ALTERNATIVE LANGUAGES FOR THE SPECTRUM

Richard Hurley & David Virgo

ALTERNATIVE LANGUAGES FOR THE SPECTRUM

Alternative Languages for the Spectrum

Richard Hurley
& David Virgo

Duckworth

First published in 1986 by
Gerald Duckworth & Co. Ltd.
The Old Piano Factory
43 Gloucester Crescent, London NW1

ISBN 0 7156 1978 0

British Library Cataloguing in Publication Data

Hurley, Richard
Alternative languages for the Spectrum.
1. Sinclair ZX Spectrum (Computer)—Programming
2. Programming languages (Electronic computers)
I. Title II. Virgo, David
001.64'24 QA76.8.S625

ISBN 0-7156-1978-0

Photoset in North Wales by
Derek Doyle & Associates, Mold, Clwyd
Printed in Great Britain by
Redwood Burn Ltd., Trowbridge

Contents

This book is dedicated to
my two daughters
Georgina and Melissa

R.H.

Preface

We would like to express our gratitude to James Mead, Graham Budd and Stephen Lacey for their valuable contributions to the chapters on micro-PROLOG and the pseudo-languages. We would also like to thank Jo Hurley for many hours spent slaving over a hot keyboard in a cold study.

All the languages described in this book are based on commercially available packages, many of which were kindly donated by their publishers: Pascal and C by Hisoft; LOGO and Micro-PROLOG by Sinclair; PILOT by Duckworth; FORTH and HURG by Melbourne House; The QUILL by Gilsoft; WHITE LIGHTNING by Oasis Software.

R.H. & D.V.

Introduction

The Spectrum, like all computers, can understand only one language directly – machine code. Unfortunately, however, since machine code consists of many instructions represented by binary numbers, it is very difficult for people to learn and manipulate. Because of this, numerous computer languages have been developed over the years, with each language being designed for some specific application. These high-level languages make use of descriptive words, which are easy for a programmer to learn, instead of numbers. The most common of these user-orientated languages is BASIC, with which all Spectrum owners will be familiar. It is easy to learn and the programs are generally clear and easy to follow, but it does have many drawbacks which can only be overcome by resorting to either machine code or some other language.

As the computer can understand only machine code (Z80 in the case of the Spectrum), any program written in an alternative language has to be translated before it can be executed. This is done by one of two possible programs, either a *compiler* or an *interpreter*, usually resident within the computer's Read Only Memory (ROM).

Whether the computer uses an interpreter or a compiler for the translation process will depend on the language and the machine. In the case of the Spectrum, BASIC is the standard language, and as BASIC is generally interpreted the machine contains a resident interpreter in ROM as shown in Fig. 1.

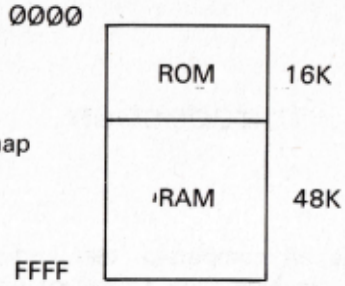
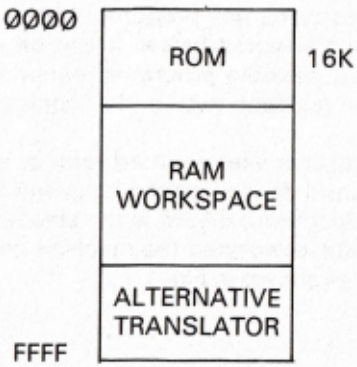


Fig. 1. Memory map

If we wish to use an alternative language such as Pascal, LOGO or FORTH on our computer, the first problem we encounter is the fact that the resident interpreter is no use as it can only be used to translate from BASIC into machine code and therefore will not understand the Pascal or LOGO commands. To overcome this problem we must load an alternative translator into memory, and as the ROM cannot be altered, this has to be stored within the Random Access Memory (RAM), either at the top or at the bottom as shown in Fig. 2.

Fig. 2. Memory map



This system is satisfactory as it allows us to program in our alternative language, which may well be faster, more efficient or more suitable for a particular application. However, it does give rise to two further problems:

1. If the computer is switched off, the contents of RAM including the alternative interpreter or compiler will be lost and

will have to be reloaded from tape or microdrive.

2. As the translator occupies RAM instead of ROM the programmer has less workspace available and this lack of memory can cause severe problems, especially in the case of a compiler-based language.

Interpreters and compilers

Although a compiler and an interpreter both do the same job, the way in which they operate is totally different, and the programmer must select which is the most appropriate.

Interpreters

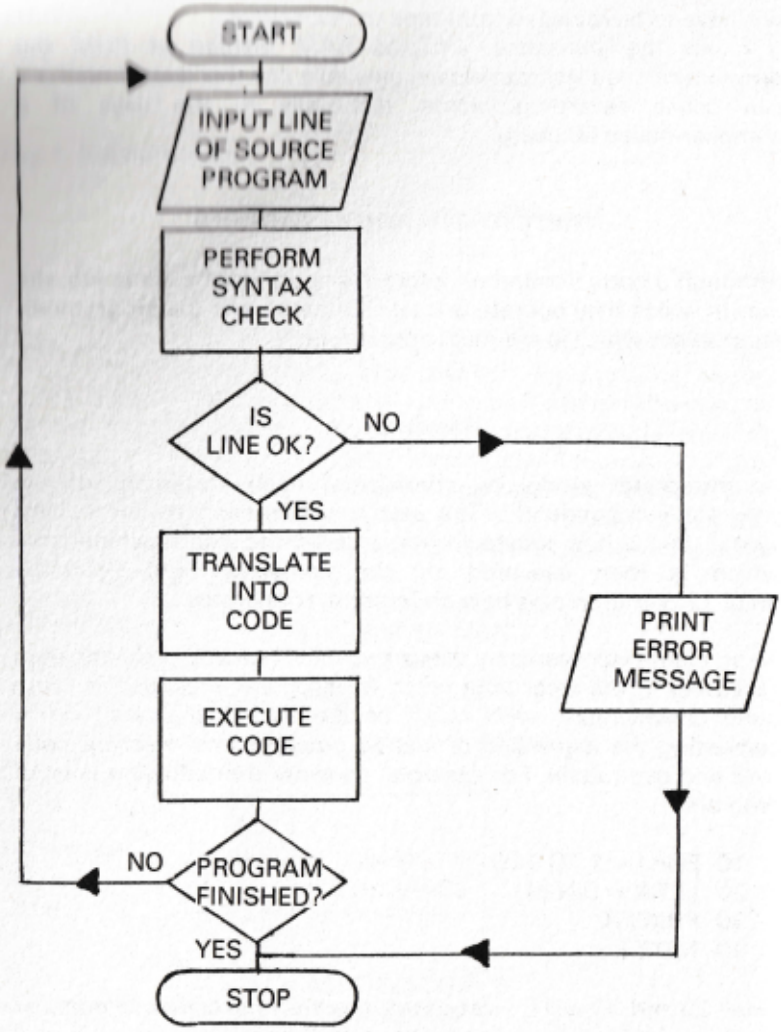
An interpreter works by considering each statement of the program independently. The statement is checked for correct syntax, and if it is satisfactory it is converted into machine code which is then executed by the computer. The flowchart on p. 12 demonstrates how an interpreter operates.

As an interpreter works by converting each line into code and then executing it, the total time taken for program execution is often quite considerable, with much of the time being used up in converting the same line of source program into machine code over and over again. For example, consider the following lines of program:

```
10 FOR I = 1 TO 100
20 LET X = SIN (I)
30 PRINT X
40 NEXT I.
```

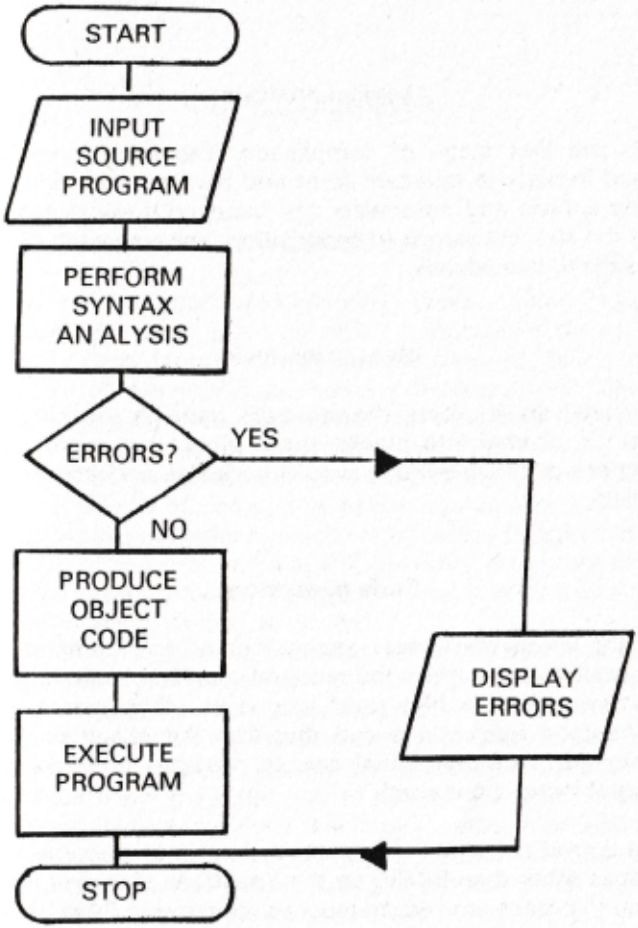
Lines 20 and 30 will be translated into machine code one hundred times, and this represents a lot of wasted time.

The main advantage of an interpreter-based system is that the program need not be complete or free from errors before it can be executed. The program will run perfectly until an incorrect or missing line is encountered, at which time some suitable error message will appear on the screen. This makes a very good environment for program development, but produces an end result which is often too slow to be of much practical use.



Compilers

A compiler, unlike an interpreter, translates the complete source program into a complete object program in machine code before it is executed. This means that when the program has been compiled it will run very much faster as there is no further translation to be carried out. The flowchart on p. 13 demonstrates the operation of a compiler.



A compiler will generally require more memory than an interpreter, as a complete source program and a complete object program are contained in the computer's memory at any one time.

Although the flowcharts shown in this introduction are short and easy to follow it should be noted that interpreters and compilers are very complex programs. The checking of individual statements and the generation of the appropriate machine code is a very complex procedure which can be broken down into three stages: *lexical analysis, syntax analysis* and *code generation*.

Lexical analysis

This is the first stage of compilation. The source program is changed into some standard form and the various redundancies such as spaces and comments are removed. Lexical analysis is one of the slowest stages in compilation, because each character is considered individually.

Syntax analysis

Syntax analysis is where the structure and the meaning of the program is divided into blocks, these blocks are separated into instructions and the various words, variables and constants are identified.

Code generation

When the lexical and syntax analyses have been completed, it is then possible to generate the required machine code. In general, one statement in a high-level language will generate several machine-code instructions and therefore the object program is often longer than the initial source program in terms of the number of instructions used.

In this introduction we have seen that it is possible to use languages other than BASIC on the Spectrum. However, to do so we require a translator which must be loaded into RAM before we can start programming. Several of these translator programs are available, and in the next six chapters we will consider a number of them and investigate the many facilities that they offer and the advantages they have over standard BASIC.

1

C-Language

Introduction

C is a general-purpose high-level programming language designed to be able to construct efficient, concise code while still permitting a high degree of flexibility. It is currently being used in the design of spreadsheets, word processors and database management programs, and it is the language that Digital Research have adopted for system development.

C was originally designed for the composition of operating systems, a branch of computing called 'systems programming', but it has proved powerful in much wider fields of application. It is possible to write extremely elegant, powerful programs using C, but it offers little protection to the user, and it is also possible to include bugs which cannot be located.

Most of the versions of C written for 8-bit machines contain only a subset of the language with some of the data types and functions omitted, and the Spectrum version, implemented by Hisoft, is no exception to this. The one main omission is floating point arithmetic and its associated functions, but the Hisoft version does allow the extension of direct execution. This takes advantage of the way in which this implementation of C compiles immediately, without passing through assembly language and a linker, to allow the user to execute the program with no delay. This means that program development can be very rapid.

The first program

When the compiler is loaded into the Spectrum it is common practice to check that the system is operating properly by entering a short program. The most popular one displays 'Hello world' on the screen; a friendly enough message, obviously designed to make you feel at home with the new language!

```
main ( )
{
   printf("Hello world\n");
}
```

The word 'main' says that this is a main program which can be executed by itself. The opening brace '{' marks the beginning of the program and the closing brace '}' marks its end in a very similar way to BEGIN and END in Pascal. Indeed, C has many similarities with other compiled high-level languages, and we will see more of these as this chapter progresses.

The code contained between the braces is therefore the main program and it should be no surprise that the 'printf' command displays information to the screen. The code \n which is contained within the quotation marks stands for new line, causing the display to select the start of a new line. This is similar to WRITELN in Pascal. Although it consists of two characters (the symbol shift D and n) it compiles to only one character in the eventual program.

It should be noted at this point that all C programs are written in lower case and that, like Pascal, all spaces are ignored so as to allow the user to display his text in as readable a format as possible. Also note the semicolon at the end of the printf line. Can you think of another language that uses semicolons as separators? Guessed it in one – Pascal!

Although all the examples included in this chapter will function correctly on the Hisoft implementation of C, it is not the prime concern of this book to explain how to use that version and so we will not be considering how to use the editor, etc. – this is clearly explained in the manual.

Let us now turn our attention to variable types and the ways in which variables can be displayed on the screen.

Variables

The memory of a computer is subdivided into bytes, each of which can contain a small number. The C compiler can associate a name (known as a variable since its contents can change) with one or more of these bytes, and this variable can be one of several different types:

char	–	a character which needs 1 byte.
short	–	a short integer which requires 2 bytes.

long – a long integer using 4 bytes.
int – a general integer which requires a machine-dependent number of bytes.

There are also three data types (float, double, range) which are used with floating point real numbers, but since the current Spectrum implementation of C does not handle these we will not consider them here.

Since a byte is 8 bits, on a standard implementation a char can hold a number in the range −128 to 127, a short can range from −32768 to 32767 and a long can hold values between −2147483648 to 2147483647. It is also possible to have a variable of type unsigned short, unsigned long or unsigned char, which can hold a number in the range 0 to 65535, 0 to 4294967295 and 0 to 255 respectively.

However, these values exist only on full size implementations; the Hisoft version only has the facilities to use 1 byte for chars and 2 bytes for short, long and int. So the range of integers which can be used is simply the same as that given for short above, independent of the type specification. It is hoped that in future versions of C a full range of types will be possible, but for now int, short and long are all identical.

In a similar way to Pascal, all variables to be used must be declared and their type must be specified at the start of each program. Here are some declarations:

```
char a,b,c ;
short i ;
long total ;
int j,k ;
```

The first of these demonstrates a notational convenience. Instead of writing

```
char a;
char b;
char c;
```

the three declarations can be linked together with the variables separated by commas.

To assign a value to a variable the C assignment statement is used. This is a simple equality sign (as in BASIC) and examples of its use are:

```
a = 'Z' ;
i = 19 ;
j = -3 ;
k = j ;
```

This should be read as "a is given the value Z", etc., *not* as "a = Z".

Constants and #define

C programs can also contain constants whose values do not change during the execution of the program, and these can be of any of the types described above for variables.

Constants can be used to make C programs easier to read, and we use the #define command to give them values. For example, the first program could be written

```
main()
#define GREETING "Hello world\n"
 {
  printf(GREETING);
 }
```

Before the program begins we tell the compiler that any time it encounters the word GREETING, it should replace this by "Hello world\n", so at compilation the printf statement is read as

```
printf("Hello world\n");
```

Constants of type integer could be defined in a similar way:

```
#define MAXINT 32767
```

Notice that the constants have been defined using upper case letters. This is a general occurrence in C programs and is done so that constants are easily identifiable within the code.

The #define command comes before the main program and the statement does not require a semicolon after it. This is because it is not part of the C program but rather a directive to the compiler to replace any occurrence of GREETING by "Hello world\n". This is signified by the #.

Binary, octal, hexadecimal and decimal

Any item of data is stored inside the computer by means of a series of 1s and 0s (a series of two state devices which are either on (1) or off (0)). So one can regard a machine code program as being a long list of binary numbers. It is convenient to express these in either *octal* (base 8) or *hexadecimal* (base 16), because this makes them slightly more readable than a list of 1s and 0s but still able to be converted quickly into the original binary.

Decimal	Binary	Octal	Hexadecimal
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

In C any number starting with a digit 0 is interpreted as an octal number and any prefixed by 0x is interpreted as a hexadecimal number. Any variable or constant can be given a value in any one of the bases; octal, decimal or hexadecimal, and can also display its value in any base.

In decimal or base 10 (sometimes known as denary) a number can be thought of as a certain multiple of units, tens, hundreds, etc. so that the values of the positions for the digits are powers of ten. For example,

$$321 = 3\times10^2 + 2\times10^1 + 1\times10^0 = 300+20+1$$

In octal these values are powers of 8, in hexadecimal they are powers of 16. With a base 16 number we need symbols to represent the decimal numbers 10, 11, 12, 13, 14 and 15: the characters A,B,C,D,E and F are employed.

Octal $0237 = 2\times8^2 + 3\times8^1 + 7\times8^0 = 128+24+7=159$
Hex $0\times B7 = 11\times16^1 + 7\times16^0 = 176+7=183$

Atoms

Whenever a C compiler converts a program into machine code, its first task is to split the code into basic pieces called atoms. Here is a complete list of atoms.

1. Names or identifiers. These are chosen by the programmer and can be of any length although it is normal for the compiler to ignore any characters after the first eight. They must start with a letter, but apart from this restriction any characters can be used to allow variables to be given meaningful names. For example,

a,t3,fine,total, supercalifragilisticexpialidocious are all legal.

2. Constants. These can be in any number base mentioned above or of type char.

3. Whitespace. Any number of blanks, tabs and new lines may be used since they are ignored at compilation time; these are known as whitespace. It is sensible to use indentation and new lines to make the program comprehensible and easy to follow.

4. Comments. These should always be included in any program to explain the operation of pieces of code. In C, comments are enclosed within the symbols '/*' and '*/'. See some of the following programs for examples of this.

5. Separators. These are C's punctuation marks:

, ; { } = ( ) : [ ]

6. Operators such as plus, minus, times, divide, etc. are explained below. The following symbols are used:

+ − * / % =

7. Keywords. These are the reserved words of the language, such as char, int, short, long, etc.

Output

We are now in a position to write a simple program involving variables, but have yet to understand the full capacity of the printf command. We have seen how to print a string constant such as 'Hello world', but to display the value of a variable we first have to say which format we require:

%d	signed decimal integer
%o	unsigned octal integer
%x	unsigned hexadecimal integer
%u	unsigned decimal integer
%c	a single character

A line such as

```
printf("%d",i);
```

will display the value of i as a signed decimal number. It is also possible to mix text with values in a printf statement, so if i is the score obtained in a test we could write

```
printf("score = %d points",i);
```

When this line is executed the %d is replaced by the value of i, so if i is given the value 60, the output is

score = 60 points

C also allows several outputs of variables in different formats in one printf instruction. Consider the following short program:

```
main()
{
    int i;
    i = 60;
    printf("value is %d or %o or %x or %u or
%c",i,i,i,i,i);
}
```

This displays the value of i in the five different formats given above, and the output looks like this:

value is 60 or 74 or 3C or 60 or <

where 60 in decimal is the same as 74 in octal, 3C in hexadecimal, 60 as an unsigned integer and is the ASCII code for the character <. Exactly the same can be done with characters. If the first two lines of the program are amended to

```
char i;
i = ',';
```

the program will produce the following output:

value is 44 or 54 or 2C or 44 or ,

This is because the ASCII code for a comma is 00101100 which is the same as 44 in decimal, 54 in octal and 2C in hexadecimal.

The format commands can be further extended to define a field length. This is similar to the Pascal field, but remember that currently we are only using integers, so the most we can do is specify the width of the field as there is no meaning to the position of a decimal point! This is achieved by placing the length of the field between the % and the character defining the format. For example,

```
printf("%6d",i);
```

When a field width is specified the character always appears at the extreme right, i.e. it is right justified. To alter this a negative sign can be placed in front of the field width and then the output is

left justified. To see how this works, amend the printf statement in the above program (with i as an integer, holding the value 60) as follows:

```
printf("value is %6d.  nvalue is %-6d." ,i,i);
```

The output will be

```
value is     60.
value is 60
```

There are other options with field types, such as filling the empty spaces with Os. These are summarised on p. 55 of the Hisoft manual.

Arithmetic

Since it is the aim of C to stay as close to assembly code as possible without making the language machine-dependent, most of the operations which can be performed on registers are readily available directly within the language. There are five basic arithmetic operations:

+	addition
−	subtraction
*	multiplication
/	division
%	remainder

It is possible to use a single minus sign '−' in front of a number or variable to indicate that it should be multiplied by −1, and brackets can be used to affect the order in which multiple operations are performed. Operations are dealt with in the usual order of precedence, which can be summarised as follows:

()	parenthesis
−	single minus
*/%	multiplicative
+	additive
=	assignment

Consider the following short program:

```
main()
{
   int i,j,k,l;
   i=2;
   j=7;
   k=8;
   l=i+j+k; /* alter this line */
   printf("answer is %d" ,l);
}
```

This will produce the output

answer is 17

namely 2+7+8. Now alter the line to read l=i+j*k;. The result should be 58 since the multiplication is performed first. If the line is further amended to l=i*–j the answer will be the result of 2 times negative 7, i.e. –14, since the single minus has the highest precedence.

Different combinations of these operations could be tried since the expression can be as long as we wish. Try to predict the value of the result for the following statement before entering it into the program to check your answer.

l=((i+j)*–k+j)%j

If you want the answer, it's at the end of this chapter.

Flow of control

Like all computing languages, C offers the programmer the facility to branch from one part of a program to another so that the program can make decisions and perform loops. The standard four branches exist, although the syntax of the code is unique to C:

if (condition) <statement> else <statement>
for (<starting point>;<finishing condition>;<increment>)<statement>
while (condition) <statement>
do <statement> while (expr)

To use any of these we need to know the relational operators which are used to test values. These are:

<=	less than or equal to
<	less than
>=	greater than or equal to
>	greater than
==	equal to
!=	not equal to

The precedence of these operators is just below the arithmetic operators, which allows us to write expressions such as i+j>=i–k, where the arithmetic is performed first.

Loops

As an example of using loops, here is a program to display the ASCII character set on the screen:

```
main()
{
  char c;
  for(c=32; c<127; c=c+1)
     printf("%c";c);
}
```

This uses the 'for' loop which is similar to the same command in BASIC and Pascal. In the brackets three conditions are given. The first is the starting value, the second is a condition for exit and the third is the increment after each repetition of the loop. The following 'for' statement in C

```
for(i=1;i<=20;i=i+3)
```

is the same as the following statement in BASIC:

```
FOR I = 1 TO 20 STEP 3
```

If more than one statement is required within the loop then the body of the 'for' statement must be enclosed between braces {} in a similar way to BEGIN and END in Pascal.

The same program could be written using a 'while' loop as follows:

```
main()
 {
  char c;
  c=32;
  while  (c!=127)
   {
    printf("%c",c);
    c=c+1;
   }
 }
```

Here the value of c is set to 32 and the loop inside the braces i repeated all the time c does not equal 127. The test is performe before the statement is executed in the same way as the 'fo statement. The loop which tests its control variable after th execution of the statement is the 'do-while' loop. This is th equivalent of the REPEAT-UNTIL in Pascal, which always execute at least once.

The program using this loop to display the same character runs as follows:

```
main()
 {
  char c;
  c=32;
  do
   {
    printf(%c",c);
    c=c+1;
   }
  while(c!=127);
 }
```

The 'do-while' command should be used sparingly because it ca lead to division by zero errors when the initial loop is performed.

Conditional execution

The 'if' statement is the simplest control statement in C. A example of its use is as follows:

```
if(i=6) printf("six");
```

where the word six is displayed on the screen if i = 6. This can b extended to the 'if-else' command where, depending on th

condition, the computer follows one of two possible paths. For example,

```
if (i=6)
    printf("six");
else
    printf("not six");
```

If the value of i is 6 then the message 'six' is displayed, otherwise the message 'not six' is shown. As with the 'while' loops considered above, the statements can be compound and in this case would be enclosed by braces {}. It is also quite legal to follow an 'else' or an 'if' by another 'if-else' command, and this process is known as nesting. It should be undertaken with care, always ensuring that the correct 'else' ends up attached to the correct 'if'!

Logical operators

We frequently need to combine the results of relational operators, i.e. we may require the computer to display the message 'origin' if x=0 and y=0. This can be achieved by using logical operators.

In Pascal we will meet these as AND, OR and NOT. The same three exist in C, but the symbols used for them are &&, || and ! respectively. So to convert the above requirement into C, we get:

```
if(x==0 && y==0) printf("origin");
```

Notice that no parentheses are required around the x==0 and y==0 since the precedence of the logical operators is below that of the relational operators.

The 'and' and 'or' operators work from left to right, and as soon as the result is determined the remaining conditions are not performed. Thus in the origin example given above, if x was not zero then the second condition would not be evaluated since the whole condition is clearly false. Similarly with the 'axis' example below, if x=0 then the result is clearly true and so the equality of y and 0 is not checked.

```
if(x==0 || y==0) printf("axis");
```

Since a false result gives a value zero (0) and a true result gives value of one (1),

!a (not a)

is the same as

a==0

Input

None of the program examples considered so far have required any input values to be specified by the user when the program is being executed. However, it is very often necessary to allow for this, and it forms an integral part of writing versatile programs. To obtain a character from the keyboard the 'getchar' command is used. This causes the computer to wait until one character is entered. If more than one is entered, the computer accepts the first character. To give a value to c, the command is as follows:

c = getchar();

Since this only accepts a character as an input, to convert this into a digit requires a little work. The ASCII code of 0 is 48 and 1 is 49 etc. So to convert the character into an integer we must remove 48. This can be done by the following program.

```
main()
/* converts a character into a digit
 */
 {
  char c;
  int i;
  c=getchar();
  if(c>='0' && c<='9')
   {
    i=c-'0';
    printf("integer is %d",i);
   }
 }
```

Another method of displaying a character is to use the 'putchar' command, which is the opposite of getchar. The print statement above could be replaced by

```
printf("integer is ");
putchar(c);
```

to produce the same output.

Lvalue and rvalue

In C, as in most high-level languages, the left-hand side of an assignment statement must be an lvalue, i.e. an expression that indicates a storage location. So in the statement 'a = b', a is an lvalue since it refers to the storage location of a.

On the right-hand side of the assignment statement we can have an lvalue or an rvalue. The latter is best defined as any expression which is not an lvalue, i.e. a constant. Thus 'x = 0' has an rvalue on the right-hand side, while our original example, 'a = b', has an lvalue.

Increment and decrement

In the examples involving loops given above, the command 'i = i+1' was frequently employed. This increases the value of i by 1. Because C is concerned with the production of efficient code, it extends its set of operators to include some special ones which have the effect of increasing or decreasing values. These operators can only be applied to lvalue operands, but they can be applied either before the operand (when they are known as prefix operators) or after it (when they are known as postfix operators). Prefix and postfix operators have quite different effects.

As an example, consider the value of b to be 42. If the increment operator '++' is applied as follows:

```
a = ++b
```

a and b will both hold the value 43. However, if the value of b is 42 and the following statement is issued:

```
a = b++
```

a will hold the value 42 and b the value 43.

The decrement operator '--' works in exactly the same way, but the values are decreased by one instead of being increased. Thus if b holds the value 42

```
a = --b
```

will produce 41 in both a and b, while the statement

```
a = b--
```

will produce 42 in a and 41 in b.

The precedence of these operators is higher than all those considered so far with the exception of parentheses. So it is quite legal to write

```
a = b*c--;
```

although a neater form of this is

```
a = b*c;
--c;
```

Because of this, more efficient code is produced when loops are used if the 'for' statement reads

```
for(c=32;c<127;++c);
```

rather than

```
for(c=32;c<127;c=c+1);
```

Bitwise logical operators

We have already noted that the final representation of any integer can be regarded as binary, and that in the Hisoft implementation any integer is stored using 2 bytes (16 bits). The bitwise logical operators have an immediate effect on these bits and are completely different from the logical operators we have met before. There are four such operators, and (&), or (|), not (~) and exclusive-or (↑), and their effects are as follows:

bit-and

AND	0	1
0	0	0
1	0	1

bit-or

OR	0	1
0	0	1
1	1	1

bit-negative

NOT	
0	1
1	0

exclusive-or

EXCLUSIVE OR	0	1
0	0	1
1	1	0

These operators are applied to the bits in corresponding posi in the two binary numbers involved in an operation. To a confusion with the logical operators &&, || and !, these bit logical operators are called bit-and, bit-or, bit-negative exclusive-or.

As can be seen from the table, the bit-negative is the sim since it has only one operand and its effect is to replace all by ones and all ones by zeros. So if x = 0x4B75 (remember t prefix 0x means that the number is in hexadecimal) we have

x = 0100101101110101 (0x4B75)
~x= 1011010010001010 (0xB48A)

This can be demonstrated by the following program:

```
main()
{
  int i;
  i=0x4B75;
  printf("%x negated is %x",i,~i);
}
```

The other bitwise operators require two operands and the eff these can be seen by the following examples, in which i ha value 0xFC77 and j has the value 0x7384.

The bit-and (&) places a 1 bit in each position where bot and j have a 1 bit, with 0s in all other positions. For example,

```
  i   1111110001110111 = 0xFC77
  j   0111001110000100 = 0x7384
i&j   0111000000000100 = 0x7004
```

The bit-or, (|) places a 1 bit in each position where either i or j has a 1 bit with 0s in all other positions. For example,

```
i|j   1111111111110111 = 0xFFF7
```

The exclusive-or (↑) places a 1 bit in any positions where i and j have opposite bits (i.e. if i=1 and j=0, or i=0 and j=1), the remaining positions containing 0s. For example,

```
i↑j   1000111111110011 = 0x8FF3
```

The precedence of bitwise operators is very confusing, so it is always a good idea to use parentheses in expressions involving more than one operation. To demonstrate the results given above, a program like the following could be used, with the line containing the operation being amended to show the different operations.

```
main()
 {
  int i,j,k;
  i=0FC77
  j=0x7384
  k=i&j; /* alter this line */
  printf("result is %x";k);
 }
```

Shift operators

The shift operators also have a direct effect on the bits which go to make up a number. The left shift operator << requires two integer operands, the first specifying the number to be operated on and the second indicating how many places the bits should be shifted. Consider the following statement:

```
0x13<<2
```

This means 'take the bits in the number 0x13 and shift them two places to the left'.

```
    0x13   =   0000000000010011
0x13<<2    =   0000000001001100
```

So the answer is 0x4C or 76. Note that the spaces at the right are filled with 0s.

Everyone knows that if we shift a decimal number one place to the left, this is the same as multiplying by a power of ten. For example, take the number 5. Shift it to the left once and you get 50, which equals 5×10^1; shift to the left twice and you get 500, which equals 5×10^2, etc.

The same is true for binary numbers. Shifting to the left is the same as multiplying by a power of 2. Thus in the above example our number 0x13, which is 19 in decimal, has been multiplied by a factor of 2^2, i.e. 4 ($4 \times 19 = 76$).

The right shift works in a similar way but shifts all the bits to the right. This can be compared with division by a power of 2. For example,

```
    0x13   =   0000000000010011
0x13>>2    =   0000000000000100
```

with the extra spaces at the left being filled with 0s. However, we have lost some information because the first two bits have been shifted out of the number.

Note, therefore, that care is needed when shifts are employed because bits can be lost at one of the two ends of a number depending on the direction of the shift.

Assignment operators

There exists in C another group of operators, known as assignment operators. These combine an assignment statement (see p. 17) with either an arithmetic or a bitwise operator. A full list of them is as follows:

i+=j	add j to i
i-=j	subtract j from i
i*=j	multiply i by j
i/=j	divide i by j
i%=j	i takes the value of the remainder when i is divided by j
i<<=j	shift the bits of i to the left j places
i>>=j	shift the bits of i to the right j places

i&=j	i takes the value of i bit-and-ed with j
i|=j	i takes the value of i bit-or-ed with j
i↑=j	i takes the value of exclusive-or of i and j.

Each of these produces shorter and more efficient code than its equivalent. These equivalent codes are the only sort which languages such as BASIC and Pascal allow. For example,

i+=j is equivalent to i=i+j
i*=j is equivalent to i=i*j
i<<=j is equivalent to i=i<<j.

Parentheses can be used, but it is normally neater to write any complex expression on more than one line. Thus

```
a=(b += (c *= d));
```

is best written as

```
c *= d;
b += c;
a = b;
```

These operations take the same precedence as the assignment statement.

Arrays

An array is a data structure which allows several items of data to be stored using a single variable name, with each individual item accessible by a reference number. These behave in exactly the same way as BASIC and Pascal arrays and are declared at the start of the main program with the other variables. For example,

```
char word [20];
```

This will set aside enough memory to store twenty characters referenced by word[0], word[1], word[2], ..., word[19]. This technique is known as 'zero origin subscripting', since the reference has an initial value of 0 (unlike BASIC where the initial value is 1).

Since it is only possible to store single characters in a variable in C, an array is a method of storing a string or word. The

following program asks you to enter your name and then replies with the message 'Hello <name>'.

```
main()
{
/* how to enter a word */
 char word[20];
 int i,j,c;
 i=0;
 printf("name = ");
 while((c=getchar())!= '\n')
 if (i<=20)
  {
    word[i] =c;
    i++;
  }
 if (i>20) i=20;
 printf("\n Hello ");
 for(j=0; j<i; ++j)
    printf("%c",word[j]);
 printf("\n");
}
```

This will allow a word of up to 20 characters, and although it will allow you to enter more, only the first 20 will count towards the name. The line

```
while((c=getchar()) !='\n')
```

is an example of the embedded assignment. This line combines two steps: first c is given a value from the keyboard using the getchar() command, and then this value is compared with the new line character. If it is equal then the loop ceases, otherwise it continues. The extra parenthesis is required since assignment is of a lower order of precedence than !=.

Conditional operator

The final operator that we will consider is the conditional operator, '?'. The conditional operator performs the same task as an 'if else' statement, but can sometimes make the code more efficient. It enables a decision to be made and then one of two actions to be performed. For example,

```
a?b:c
```

If a is true (not zero) then b is performed, otherwise a is false (zero) and c is performed. Here is an example of its use which places into the variable absx the absolute value of x:

```
absx = x<0 ? -x:x
```

The condition is 'x<0'. If this is true then absx = −x, otherwise absx = x. It is the same as

```
if (x<0)
 absx = -x;
else
 absx = x;
```

Summary of precedence

Here is a table to show the order of precedence of operators in C, with the most important at the top. Operators that have equal precedence appear on the same line.

Operator type	Operators
Primary	( ) [ ]
Monadic	& ! − ++ −−
Arithmetic	* / %
	+ −
Shift	>> <<
Relational	< <= >= >
	== =
Bitwise	&
	↑
	|
Logical	&&
	||
Conditional	?
Assignment	= += −= %= /= |= ^= &= >>= <<=

Character types

The standard C library contains a set of built-in functions which test for the type of character which a char variable holds. In the example program given above on p. 28, which converts an input character into its integer equivalent, you will find the line

```
if(c>='0' && c<='9')
```

This is a test to determine if c holds a character in the range 0-9, i.e. a digit. The character type test for this allows the line to be replaced by

```
if( asdigit(c))
```

which returns true if c is a digit. A full list of the available tests is as follows:

isalnum(c)	is c a letter or a digit (alphanumeric)?
isalpha(c)	is c a letter?
isascii(c)	is c an ASCII character (code less than 0x80)?
iscntrl(c)	is c a control character (code less than 0x20)?
isdigit(c)	is c a digit?
islower(c)	is c a lower case character?
isprint(c)	is c a printing character (including space)?
ispunct(c)	is c a punctuation mark (i.e. a printing character other than a digit or letter)?
isspace(c)	is c a whitespace character (i.e. space character, new line character or tab character)?
isupper(c)	is c an upper case character?

There are also two functions which will convert characters between upper and lower case:

tolower(c)	converts an upper case character into a lower case one.
toupper(c)	converts a lower case character into an upper case one.

Functions

This chapter has introduced you to a collection of functions which already exist in the standard C library. The way to improve your programs and thus benefit from the power of C is to develop your own functions and incorporate them into your own library. Therefore the rest of this chapter is devoted to the definition and construction of functions, along with some useful examples. But first let us ask, 'What is a function?'

A function is a set of statements performing some computation which can be called for by a reserved word, i.e. printf(), getchar(), isdigit(), etc. To define your own function, it must perform one specific task and have a unique name.

Like Pascal, Hisoft C does not include an operator for raising one number to the power of another, so if this task is to be performed we must write our own. Because we are restricted to integers, we will limit ourselves to raising positive integers to positive integral powers.

Any function will consist of the following

1. Type
2. Name
3. Parameters (optional)
4. Parameter declaration list (optional)
5. Block

Here then is the function power (x,y) to find x^y:

```
int power(x,y)
  int x;
  int y;
   {
    int i, tot;
    tot =1;
    if(x<0 || y<0)
     {
      printf("error\n");
      return(0);
     }
    else
      if(y!=0)
        for(i=1; i<=y; i++)
          tot *= x;
    return(tot);
   }
```

This should be placed at the start of the program and be followed by the main program as follows:

```
main()
{
 printf("%d",power(4,6));
}
```

This will call the function 'power' to calculate the value of 4^6. It then displays the result, 4096.

If we examine the function we can see the five parts listed above:

1. Type: this is the first word and it is 'int' since the function will return an integer answer. If there is to be no value returned then the word to use is 'void'.
2. Name: the function's name is 'power'.
3. Parameters: the parameters of the function are x the base, and y the exponent.
4. Parameter declaration list: this informs the compiler that x and y are integer variables.
5. Block: the main block comes between a pair of braces {}, and can be treated as if it were a program in its own right.

The command 'return' tells the compiler to go back to the main program, and the value following it in brackets is the value of the function. Any variables declared in the body of the function are local variables and have no effect on those variables outside in the main body.

Earlier in this chapter we wrote a program to convert a single character into its corresponding digit. Here is a function to accept up to five characters (only accepting digits) and convert these into a decimal number.

```
int number()
{
 int i,c,tot;
 char s[5];
  i=0;
  while ((c=getchar()) != '\n')
    if (isdigit(c))
      s[i++]=c;
  tot=0;
  for (c=0;c<i;++c)
    tot=tot*10 + s[c] - '0';
  return(tot);
}
```

This routine performs no test to see if the number is too large for an integer (i.e. over 32767) and only works for positive numbers. Notice that with this function there is no parameter list and hence no parameter declaration, although brackets are still needed.

We can now combine these last two functions by writing a

main program that asks for a lower limit, an upper limit and an increment before displaying a table of cubes between these limits.

```
main()
{
 int low,upp,stp,cube,i;
  printf("TABLE OF CUBES\n");
  printf("===== == =====\n");
  printf("\nlower limit = ");
  low=number();
  printf("upper limit = ");
  upp=number();
  printf("increment   = ");
  stp=number();
  printf("\n");
  printf("NUMBER  CUBE\n");
  printf("======  ====\n");
  for (i=low;i<=upp;i+=stp)
    {
     cube=power(i,3);
     printf("%4d%8d\n",i,cube);
    }
}
```

In this main program we have also used the variable i and this is the global variable i. Whenever the function number is called which also uses a variable i, this value is unaltered. Even if a function has no arguments, the brackets are still required in the function call despite there being nothing between them.

Placed together, these two functions and main program form a program of fifty lines, which when executed gives a sample run as follows (remember that the values are specified at run time):

```
TABLE OF CUBES
===== == =====

lower limit = 1
upper limit = 13
increment   = 2

NUMBER  CUBE
======  ====
     1     1
     3    27
     5   125
     7   343
     9   729
    11  1331
    13  2197
```

Conclusion

There is much more of the C language to be discovered than these pages can hold, for this is after all a reasonably short introduction. Functions are normally stored in a function library, and these can be exchanged with other users as more and more powerful functions are written. This 'toolbox' approach will lead to a self-written library of useful functions.

C is clearly a language of the future. It is fast, efficient, friendly and above all highly structured, and it is therefore definitely a language with which to become familiar.

Incidentally, the answer to the arithmetic problem set on p. 24 is 0!

2

FORTH

Introduction

FORTH is an interactive language. Programs are developed and tested at the keyboard, with the computer replying to every line as it is entered. It is a most unusual language, very different from the more common high-level languages such as Pascal and BASIC.

In the Introduction we explained the concepts of compilation and interpretation, and stated that all high-level languages use one or the other of these processes to translate programs into machine code. FORTH is different in that it uses both an interpreter and a compiler. If a normal line of FORTH is entered at the keyboard, it is interpreted and executed immediately. If, however, the same code were to be entered in the form of a colon definition then it would be compiled. In this way, FORTH offers the programmer the best of both worlds, i.e. an interactive interpreter environment for the development and testing of programs, with the finished product being compiled. This makes FORTH very fast compared with BASIC, and it is often used where speed is of great importance.

Another feature of FORTH which results in short development times is the way in which programming consists of using the existing set of words to extend the dictionary with a series of more complex words. These new words are in turn used to create even more complex operations, thus extending the vocabulary to suit the individual programmer's requirements.

Reading the above introduction may have convinced you that FORTH is the perfect language. This is not the case, however, as there are two major drawbacks from the programmer's point of view:

1. Integer arithmetic. The first and major drawback of FORTH is that we can only use integers. The reason for this is that integer arithmetic is much faster than floating point arithmetic, and the majority of applications only need integers anyway. For applications which do require real numbers in arithmetic, FORTH provides the experienced programmer with a set of double-

precision (32-bit) numbers which may be used to implement fixed-point arithmetic.

2. Reverse polish. The second drawback is that FORTH uses reverse polish as opposed to infix notation in order to represent mathematical expressions. From a computational point of view this is most satisfactory, but the newcomer to FORTH is likely to find this the biggest stumbling-block until he becomes familiar with the system.

There is little doubt that the advantages of programming in FORTH far outweigh the disadvantages, and provided that your application does not involve complex decimal arithmetic and you can master reverse polish, there are enormous benefits to be gained from using it.

Reverse polish

The first hurdle to be overcome when writing programs in FORTH is reverse polish notation. This is a method of representing arithmetical expressions, invented by a Polish mathematician called Lukaseweicz, which does not require brackets. This, together with the fact that the operators are always declared before the operation linking them, makes reverse polish notation an ideal way of representing expressions for ease of translation into machine code. The table below shows a series of standard infix expressions, together with their reverse polish equivalents.

Infix	Reverse polish
A+B	A B +
A*B	A B *
2A+B	2 A * B +
2(A+B)	A B + 2 *
2((A+B)–C)	A B + C – 2 *
3A–4B	3 A * 4 B * –

The stack

FORTH makes use of a *stack* and reverse polish notation in order to evaluate arithmetical expressions. At first this method may seem difficult and tedious, but with practice it soon becomes clear that it is far better than the usual infix notation.

If we simply type the number '27' into a computer operating in FORTH, it will reply with

```
27 OK
```

Nothing appears to have happened, even though the operating system seems to think that everything is OK. In fact something has happened; the number '27' has been placed (PUSHed) on to the stack, where it remains awaiting further instructions. We can see that it is on the stack by using the full stop symbol, '.', which has the opposite effect, removing the number from the top of the stack (this is known as popping) and printing it on to the screen.

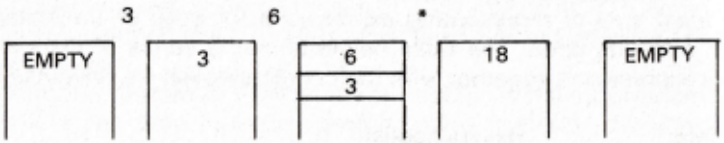
The stack can therefore be thought of as a type of notepad, on which we can write down a number to be stored for future use.

Arithmetic

Consider the problem of evaluating 3 * 6 and printing the result. To do this in FORTH we would simply type:

```
3 6 * .
```

The effect of this on the stack is shown in the diagram below:



From the diagram we see that numbers are pushed on to the stack until we reach an operator, the stack is then popped, the answer evaluated and the result pushed back on to the stack where it remains until it is printed by the full stop.

The following are examples of how certain infix expressions could be evaluated by using reverse polish and FORTH:

```
2*(A+B)          A B + * .
(A+B)*(C-D)      A B + C D - * .
2(3A+B)          A 3 * B + 2 * .
```

Numbers

As mentioned earlier, FORTH can only handle integers. However, these can be represented in several ways (as shown in the table below), and can be manipulated at great speed.

BYTE	0 to 255	1 LOCATION
SINGLE-PRECISION	–32768 to 32767	2 LOCATIONS
UNSIGNED INTEGER	0 to 65535	2 LOCATIONS
DOUBLE-PRECISION	–2147483648 to 2147483647	4 LOCATIONS

The normal single-precision and the double-precision integers are stored using the two's complement number system. Two's complement is a method of representing negative numbers in binary by using the most significant bit (MSB) as a negative value. For example,

```
0010111010100100 = 4+32+128+512+1024+2048+8192 =  11940

1000000101101110 = 2+4+8+32+64+256-32768        = -32402
```

The column headings are all positive except for the one on the far left.

If we have a situation in which we are not going to require the use of negative numbers, then we can switch from the normal default twos complement system to one in which the MSB is positive, taking a value of 65536 (2^{15}). This does not increase the overall range of numbers but does allow the use of higher positive values.

If we are using unsigned integers, the operation '.' will not print the correct value if the number exceeds 32767. For example,

```
3000 3000 + . --> (the wrong answer)
```

it is necessary to inform the computer that the number to be read from the stack is an unsigned integer, and we do this by using the operator 'U.'. For example,

```
3000 3000 + U. --> 5000 (the correct answer).
```

Double precision

Very large values are represented as double-precision numbers. And as these have to be represented on the stack in a different manner (4 bytes instead of 2), FORTH provides the user with an alternative set of operators which must be used:

- D+ Add the two double-precision numbers on the stack, leaving the result as a double-precision number on top of the stack.
- D+- Apply the sign of the number on the top of the stack to the double-precision number underneath, leaving the result as a double-precision number on the top of the stack.
- D. Displays the double-precision value from the top of the stack.
- DABS Leaves the absolute value of a double-precision number on the top of the stack.

Before a number can be operated on using these double-precision operators, it needs to be saved on the stack in the correct format. This is achieved by terminating all double-precision numbers with a 'point'. For example,

```
27. 74613. D+ D. --> 74640 OK
```

As well as the words given above, which work entirely with double-precision numbers, there are others which involve double-precision numbers during the operation:

- M* This multiplies two single-precision numbers together, storing the result as a double-precision number.
- U* Similar to the above, except that all numbers are considered to be unsigned.
- U/MOD This requires an unsigned double-precision dividend which is divided by a single precision divisor giving a result and a carry as unsigned single-precision integers.

```
724 1076 M* D.          --> 779024      OK
2176 1047 U* D.         --> 2278272     OK
742063. 127 U/MOD . .   --> 5843  2     OK
```

As you can see, although we are limited to using integers, these can be manipulated very quickly and efficiently. By using some complex techniques it is possible to perform floating point arithmetic, but that is beyond the scope of this book.

Division

The use of integers is only a problem in the case of division, where dividing one integer by another often yields a non-integer result. To cater for this, FORTH offers two types of division:

1. The operator '/' gives the integer part of a division. For example,

```
11 3 / .  --> 3 OK
```

2. The operator '/MOD' gives the answer as well as the quotient (putting both values on the stack). For example,

```
11 3 /MOD . .    --> 3 2 OK
```

It is also possible to obtain the remainder only by using the operator 'MOD'. For example,

```
11 3 MOD .  --> 2 OK
```

MAX and MIN

The FORTH vocabulary contains two other very useful arithmetic operations, namely 'MAX' and 'MIN'. Both these operators require two operands on the top of the stack, and the result (the smallest or largest) is then left on the stack. For example,

```
10 20 MAX .  --> 20 OK
-5  4 MIN .  --> -5 OK
```

Sign changes

Within FORTH there are two operators which can be used to

affect the sign of an operand:

1. ABS has the effect of removing the sign (finding the absolute value) of the number on the top of the stack. For example,

```
-4 ABS .  --> 4 OK
```

2. MINUS affects all numbers (positive and negative) and changes the sign. For example,

```
-6 MINUS .  --> 6  OK
 7 MINUS .  --> -7 OK
```

Stack manipulation

The stack is the most important factor in FORTH, so its manipulation is of extreme importance. The commands listed and demonstrated below, supplied within the FORTH vocabulary, are all concerned with stack manipulation.

DUP

The DUP operation can be used to duplicate what is at the top of the stack at any time. This is very useful if we wish to obtain an intermediate print-out without changing the contents of the stack. For example, consider the problem of calculating and printing the cumulative total of 3, 6, 9. We could use

```
3 6 + DUP . 9 + .  --> 9 18 OK
```

At this point it is useful to introduce the operation CR, which prints a carriage return/line feed on to the output device. A better presentation of the above example could be obtained by using

```
3 6 + DUP . CR 9 + . --> 9
                         18 OK
```

OVER

OVER is a quick method of duplicating the second number in a stack. For example,

```
3 6 OVER . . .  --> 3 6 3 OK
```

SWAP

This exchanges the top two numbers on the stack. For example,

```
4 6 SWAP . .    --> 4 6 OK
```

ROT

This rotates the top three numbers on the stack, making the third become the first. For example,

```
4 6 7 ROT . . . --> 4 7 6 OK
```

The stack manipulation operations considered here can be very useful in overcoming the difficulties which arise from the handling of a LIFO (last in, first out) list data structure.

Variables

As in Pascal and other high-level structured languages, all variables in FORTH must be defined before they can be used. This is achieved by using the word 'VARIABLE'. For example,

```
0 VARIABLE A  --> OK
```

In this case, a memory location for the variable 'A' will be reserved, after which it can be referred to by name.

On the Spectrum, as with most versions of FORTH, there is no restriction on the length of a variable name, but only the first four letters will be placed in the dictionary.

STORE (!)

Once a variable has been defined it can be given a value using the STORE (!) operation. This takes the number from the top of the stack and assigns it to the appropriate variable, storing the value in the memory location reserved at the declaration stage. For example

```
4 A !
```

assigns the value 4 to the variable A.

FETCH (@)

The word '@' has the opposite effect to '!' and is used to obtain the value of the variable, which is then placed on the top of the stack, from which it can be taken for further operations. For Example,

```
A @ . --> 4 OK
```

Because the word '@' and '.' are used together on numerous occasions, there exists a special operation, '?', which has the effect of both, but saves both time and valuable memory. For example,

```
A ? --> 4 OK
```

Using the FETCH and STORE commands together with the stack may seem somewhat complicated at first, but with practice it soon becomes easy.

BASIC	FORTH
LET A=A+1	A @ 1 + A !
LET X=2*Y+W	Y @ 2 * W @ + X !
LET K=2(J+1)	J @ 1 + 2 * K !

Constants

Constants are convenient ways of representing common values by a meaningful name, rather than having to state the value each time. Like variables, any constant used in FORTH must be declared using the reserved word CONSTANT. For example,

```
214 CONSTANT LENGTH --> OK
```

defines a constant called 'LENGTH' with a value of 214, and this value can be placed on to the stack at any time using the name 'LENGTH'.

The operations CONSTANT and VARIABLE are defining words,

which add new entries to the dictionary. In the case of a constant, the assigned value is fixed at the time when the constant is defined, and cannot be changed in the same way as a variable.

If we need to change the value of a constant then it is necessary to re-define it with a new value. For example,

```
121 CONSTANT LENGTH --> OK
 67 CONSTANT LENGTH --> OK
```

After these two lines have been entered, both entries will exist in the dictionary. However, as the dictionary is always scanned in reverse, the new definition will always be found. If for some reason we wish to revert to the original value of the constant, then this could be achieved by using the housekeeping operation (FORGET), which will erase the most recent definition from the dictionary. For example,

```
FORGET LENGTH
```

Colon definition

So far we have considered two defining words, VARIABLE and CONSTANT, which have the affect of adding new words to the dictionary. The FORTH word COLON, ':', is also a defining word, but is more flexible than the others. Using it we can add new words to the dictionary and define the action that will be taken when the word is used. The syntax of a colon definition is as shown below:

```
: AREA   DUP * 3 *   ;
```

We see that a colon definition consists of a <name> and the body (or action to be taken), enclosed between a colon ':' and a semi-colon ';'. The body of the definition may be any valid sequence of FORTH words, and this is compiled into the new dictionary entry.

When a colon definition has been executed, we have at our disposal a new FORTH word which can be used in the normal manner. For example, the word 'AREA' defined in the above example could be used as follows:

```
6 AREA .  --> 108 OK
2 AREA .  -->  12 OK
```

We can therefore calculate the area of any circle, simply by stacking the radius and using the newly defined word 'AREA'.

Like the name of a variable or constant, the <name> part of a colon definition can be of any length and consist of any characters. However, in many versions of FORTH, such as that on Spectrum, it is important to make the first three or four characters unique.

Consider the problem of defining a second colon definition, to calculate the volume of a cylinder given its radius and length.

```
0 VARIABLE RADIUS
0 VARIABLE LENGTH
: VOLUME
RADIUS @     ( GET VALUE OF RADIUS)
AREA         ( CALCULATE AREA)
LENGTH @ *   ( MULTIPLY AREA BY LENGTH)
;
```

To use this definition to calculate the volume of a cylinder we would assign values to the two variables and call the procedure as follows:

```
14 RADIUS !
4 LENGTH !
VOLUME .
```

This example demonstrates two new facilities:

1. A colon definition may occupy more than one line of input and even though we may hit enter after each line, FORTH does not complete the definition and print 'OK' until a semi-colon is encountered.

2. The program or definition can be annotated by putting comments in brackets. It must be remembered, however, that the open bracket '(' is a FORTH word and must therefore be surrounded by spaces. The close bracket ')' is not a FORTH word, but simply a delimiter, and spaces are therefore not necessary.

Printing text

In all the examples we have considered so far, the computer has solved a problem, placing the result on the top of the stack. By

using a dot, '.', this result can be output to the screen, producing a result such as:

```
24 OK
```

In many cases, especially from the end-user's point of view, such a brief answer is unsatisfactory, and in a language such as BASIC or Pascal we might produce a more detailed answer using:

```
PRINT "RESULT=";A
WRITELN('RESULT=',A);
```

In FORTH, the same effect can be produced by using (.") to print a textual message on the screen. When a (.") is encountered, all the text following the quotation mark and up to the next quotation mark will be reproduced on the screen.

For example, the following colon definition defines a word CUBE which will evaluate the cube of a number placed on the stack, printing the result in the form CUBE = 27.

```
: CUBE
DUP DUP * *
." CUBE = " .
;
```

This definition could now be employed as follows:

```
6 CUBE --> CUBE = 216
4 CUBE --> CUBE = 64
```

One drawback of an immediate language such as FORTH is that the output is often produced in a badly formatted way on the same line as the input. This problem can be overcome by using the two words 'CR' and 'SPACES'. 'CR' is used to force a carriage return (new line), and 'SPACES' can be used to emulate the useful 'TAB' function, available in BASIC.

By using these two functions, together with 'CLS', we can improve our colon-definition for finding the volume of a cylinder as follows:

```
: VOLUME
RADIUS @
AREA
LENGTH @ *
CLS CR
." VOLUME=" .
;
```

This new definition can be used in the same way as before, by setting the values for 'RADIUS' and 'LENGTH' followed by the word VOLUME. For example,

```
6 RADIUS !
8 LENGTH !
VOLUME
```

In this case the screen will be cleared and the result will be displayed in a satisfactory format in the top left-hand corner.

Interpretation and compilation

At the beginning of this chapter we stated that both interpretation and compilation are employed in the execution of FORTH programs. Now that we have investigated a number of FORTH words, we can examine these two modes of operation.

Consider the simple addition:

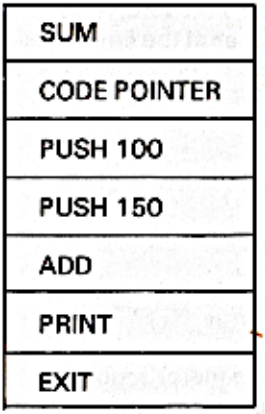
```
100 150 + . --> 250 OK
```

As soon as this has been entered, the FORTH operating system interprets the input by checking each word with the dictionary. If it is found then it is executed, otherwise it is treated as a number and pushed on to the stack.

Any FORTH entered in this manner could be included in a colon definition between the words (:) and (;). The colon has the effect of switching FORTH from the interpret mode into the compile mode, and the semi-colon at the end has the opposite effect. If we include the code in the previous example within a colon definition as follows:

```
: SUM 100 150 + . ;
```

the colon generates a new dictionary entry with the name 'SUM' and switches the operating system into the compile mode, so that the program terminating in the semi-colon is translated into individual instructions which are placed into the dictionary entry as shown on page 55:



DICTIONARY ENTRY

One very important part of the entry is the 'CODE POINTER', which points to a machine code routine which will call the various routines needed to execute the program.

In practice the dictionary is less complex than it might seem from the above description, since the addresses of the various machine code routines will be stored instead of the individual instructions.

As we have seen in the last few pages, FORTH relies on new words being defined and added to the dictionary. It is important to remember that we define these new words to be used at a later date, and it is therefore important to use short meaningful names wherever possible.

Program structures

Any reader familiar with any computer language will appreciate the need for control words, to be used in the operation of a program. Without them it would only be possible to execute the instructions of a program once each, in the order in which they were entered. This would be impractical, and FORTH allows the use of several types of control structure.

Loops

The most common control structure is a simple loop, which enables a particular section of code to be repeated a number of times. In FORTH the words 'DO' and 'LOOP' can be used for this

purpose. 'DO' stores the initial value and limit; 'LOOP' increments the value and branches back to 'DO' until the limit is reached.

For example, the following colon definition uses a LOOP to print a message on the screen ten times.

```
: TEST
11 1 DO
CR ." THIS IS A TEST"
LOOP
;
```

To produce the result we simply enter 'TEST', and the message will appear.

It is also possible to define a general loop and specify the number of times it is repeated, when we execute the word.

For example, the following defines a word called 'STARS' which will produce a line of stars on the screen.

```
: STARS
0 DO
." *"
LOOP
;
```

We now have a general loop and the number of stars, i.e. the limit of the 'DO', can be specified when the word is used. For example,

```
10 STARS --> **********OK
 4 STARS --> ****OK
```

Comparison with the FOR-NEXT loop in BASIC demonstrates the power and flexibility of FORTH, since to produce a general loop of BASIC would require the introduction of an INPUT statement.

In the BASIC version of a FOR-NEXT loop a variable is used to keep account of the current position. In FORTH, however, this is not the case, as no variable is required. If we wish to keep account of the control counter then we can use the FORTH word 'I', which will place the current value on the top of the stack. For example, to print the numbers from 1-10 we would type:

```
: COUNT 11 1 DO I . LOOP ; --> OK
```

or

```
: COUNT DO I . LOOP ; --> OK
```

then to print the numbers we would use

```
COUNT --> 1 2 3 4 5 6 7 8 9 10
```

or

```
11 1 COUNT --> 1 2 3 4 5 6 7 8 9 10
```

The second definition produces the same result, except that it is more general.

In all the loops that we have considered so far, the increment has always been 1. If we wish to choose some other value then this can be achieved by replacing the word 'LOOP' with '+LOOP'. For example, the following simple definition prints odd numbers between the limits entered at execution:

```
: ODD DO I . 2 +LOOP ; --> OK
10 1 ODD --> 1 3 5 7 9
```

IF ... THEN ... ELSE

All high-level computer languages have an IF ... THEN ... ELSE structure, and FORTH is no exception. As with most FORTH operations, the relationals (e.g. 'greater than', 'less than') test the numbers on the top of the stack, and then place a Boolean (see p. 122) value, '0' or '1', on the stack depending on the result of the comparison.

The relation tests available are as follows:

0= This tests to see if the number on the top of the stack is zero. If it is then '1' is placed on the stack, if not then zero is stacked.

0< This tests to see if the number on the top of the stack is less than zero, in which case a '1' is placed on the stack.

< This removes the top two items from the stack and leaves 'l' if the second item is less than the top.

> As above but leaves '1' if the second item is greater than the top.

It must be remembered that these only perform the test and put a Boolean value on to the stack according to the result.

```
4 5 < .    --> 1  (4 is less than 5)
3 7 > .    --> 0  (3 is not greater than 7)
-2 -6 < .  --> 0  (-2 is not less than -6)
```

Now that we have a value on the stack which is dependent on some relational check, we can use this value to produce a branch in the normal sequence of instructions. By using the words 'IF' and 'THEN' we can include a section of code between the words which will only be executed if the value on the top of the stack is TRUE (i.e. not zero). For example, if we enter the following colon definition:

```
: CHECK IF ." TRUE" THEN ." FINISHED"
```

this will check the top of the stack producing results as follows:

```
0 CHECK --> FINISHED OK
1 CHECK --> TRUE FINISHED OK
```

This system is very limited as it only allows us to skip forward if the result of some test is false. By incorporating the word 'ELSE', we can obtain greater flexibility, in that one of two courses of action can be taken. For example,

```
: TEST IF ." TRUE" ELSE ." FALSE" THEN ." STOP" ;
0 TEST --> FALSE STOP OK
1 TEST --> TRUE STOP OK
```

It is also possible to extend the number of possibilities by nesting the IF ... THEN statements. For example, the following colon definition checks the number on the top of the stack, and if it lies in the range 1 to 3 then the written equivalent is displayed.

```
: EXAMPLE
DUP DUP
1 = ." ONE" ELSE
2 = ." TWO" ELSE
3 = ." THREE"
ENDIF ENDIF ENDIF
;
```

Whenever an IF statement is evoked it must be closed by using an 'ENDIF'. This means that the number of 'IF' statements must be the same as the number of 'ENDIF's.

BEGIN ... UNTIL

The structure formed by using BEGIN and UNTIL is essentially a loop without an iteration counter. 'BEGIN' marks the beginning of the structure; 'UNTIL' tests for a true or false value on the stack. If the value is false, then control is returned to 'BEGIN'. If true, the words following UNTIL will be executed.

For example, the following colon-definition will print all the numbers in the stack, terminating on the first occurrence of zero.

```
: PRINT-STACK
BEGIN
DUP CR . 0=
UNTIL
;
```

CASE

One of the most useful commands found in high-level languages, is the CASE statement. This allows for the testing of a number against several different values. Unlike most versions of FORTH, Spectrum FORTH supports the CASE structure, which can be used to replace nested IF ... ENDIF lines, which can be very tedious and difficult to follow.

A case statement is used in the following colon-definition to produce an effect similar to that obtained in the last example. By using the CASE statement we are able to extend the range of numbers to 1-5 without introducing more lines of code.

```
: BETTER
CASE
1 OF ." ONE" ENDOF
2 OF ." TWO" ENDOF
3 OF ." THREE" ENDOF
4 OF ." FOUR" ENDOF
5 OF ." FIVE" ENDOF
ENDCASE
;
```

If we compare this with the previous example we observe that there are numerous advantages, including the fact that the number being tested need only appear on the top of the stack, i.e. it does not require duplication.

All the structures considered in this section, including LOOPs, IF ... ENDIF, BEGIN ... UNTIL and CASE, can only be used within colon definitions. They cannot be used when operating in the immediate mode.

The text editor

Consider the problem of entering a very long and complex colon definition. When it has been entered, the FORTH operating system will compile the source code and make an appropriate entry into the dictionary. The source code itself, however, is lost, and if an error occurs at run-time then the whole definition will need to be re-typed, a very tedious process.

Most implementations of the language solve the problem by dumping the source code on to disk and supplying the programmer with a minimum amount of work. As the Spectrum does not support disk drives, the problem is overcome by using the top 11K of memory as a RAM-disk. This may not seem a great deal of memory, but as FORTH is very compact a large amount of code can be stored within this area.

The RAM-disk is divided into eleven pages (numbered 0-10), with each page containing 16 lines of 64 characters per line. This is the standard FORTH format, as opposed to the Spectrum screen format normally used.

The RAM-disk

The RAM-disk is used in conjunction with the text editor to produce colon definitions, for which the source code is not lost during execution. Before using it we must prepare the RAM-disk for writing by using the command:

```
INIT-DISC
```

This simply clears the area with blanks, so that the editor can be entered using the command 'EDITOR'. We then select our page by using one of the following commands:

```
n LIST
n CLEAR
```

where 'n' is the screen number, which must be an integer in the range 0-10.

Line-editor commands

Now that we have entered the editor, we are in a position to enter, change or delete text, and these tasks are achieved using the following FORTH words:

n D	Delete the line, move lower lines up and hold a copy of the deleted line in PAD.
n E	Erase the line with blanks.
n H	Hold the contents of the line at PAD.
n I	Insert the text from PAD at the specified position.
n P	Put text onto line.
n S	Spread the text inserting a blank line.
n T	Type the contents of the line and also copy it to PAD.

In the above definitions, constant reference is made to PAD. This is the text buffer. It can hold one line of text used or saved by a line-editing command. The PAD can be used to transfer a line from one screen to another, as well as to perform edit operations within a single screen.

String editing

The screen of text being edited resides in a buffer area of storage. The editing cursor is a variable holding an offset into this buffer area. Commands are provided for the user to position the cursor, either directly or by searching for a string of buffer text, and to insert or delete text at the cursor position.

The following commands can be used to position the cursor within a page of text.

TOP	Position cursor at the start of the screen.
n M	Move the cursor by an amount 'n'.
n LIST	List screen 'n' and select it for editing.
n CLEAR	Clear screen 'n' with blanks and select it for editing.
n1 n2 COPY	Copy screen n1 to screen n2.
L	List the current screen. The cursor line is

relisted after the screen listing to show the cursor position.

F text — Search forward from the current position until string "text" is found. The cursor is left at the end of the text string, and the cursor line is printed. If the string is not found an error message is given and the cursor is repositioned at the top of the screen.

B — Used after F to back up the cursor by the length of the most recent text.

When a definition has been satisfactorily completed, we can exit the editor by typing LIST or CLEAR. The advantages of using this method of forming the colon definitions are as follows:

1. The source code is not destroyed at the time of compilation and can therefore be changed and re-compiled at any time.

2. The definitions contained within the RAM-disk can be saved and loaded by using the commands:

```
LOADT
SAVET
```

Input/output

So far we have used '.' and '."' for text output and input, relying on the fact that any number entered will automatically be placed on to the stack. FORTH does, however, provide an additional set of input/output operations which can be used together to solve most problems.

EMIT

This is the simplest output operation, which can be used to output the character whose ASCII code is on top of the stack.

```
66 EMIT --> B OK
```

The main use of 'EMIT' is for printing control characters which cannot be output in the normal manner by using the '." ' operation.

WORD

WORD has the effect of copying characters from the input stream into a 'word buffer', up to a delimiter character. The number of characters copied is held at the front of the word buffer. Note that WORD can only be used within a colon definition, as shown below:

```
: COPY 10 WORD COUNT TYPE ;
```

The word 'COUNT' simply fetches the byte pointed to by the top of the stack, and the word 'TYPE' is used to print the string whose start address and character count are on the stack.

This example is somewhat trivial. Although we are able to input and then output a string, this is achieved in a single operation, and the string cannot be recalled at a later date. If we want to be able to recall a string at any time, then it must be placed into memory at some pre-defined position. This can be achieved by defining two colon definitions, 'PUT' and 'GET', as shown below:

```
0 VARIABLE STRING 20 ALLOT
: PUT
STRING 20 EXPECT
;
: GET
STRING 20 -TRAILING TYPE
;
```

The word 'PUT' can now be used to store a string which can be recalled at any time using the word 'GET'.

These two definitions have made use of several new words, which are defined as follows:

ALLOT

'ALLOT' is used in conjunction with 'VARIABLE' to reserve a section of memory for the most recently defined dictionary entry. For example, the line

```
0 VARIABLE HELLO 20 ALLOT
```

will make a dictionary entry called HELLO which will contain 20 empt', bytes within its parameter field. The address of the first byte will be placed on top of the stack whenever HELLO is executed. We see from this example that we have reserved a portion of memory for a collection of data items, which are referred to by the name HELLO. This is similar to the DIM statement used in BASIC:

```
20 DIM HELLO (20)
```

EXPECT <address> <count>

This transfers characters from the input buffer to the address stipulated until a return or the count of characters has been received.

–TRAILING

This is used in the output stage to suppress the output of trailing blanks. Without this, twenty characters would be printed no matter how long the input string.

From the above examples and definitions we see that the input and output of strings is quite complex in FORTH, and we have to use processes similar to those employed in Pascal. This section only demonstrates one method of handling strings; others exist and these should be investigated by reading a more detailed description of the language (see Bibliography).

Conclusion

In this chapter we have barely scratched the surface of this vast and useful language. It is the enormous number of available operations which makes FORTH so powerful in the construction of programs that require compact and fast-executing code. One of the many applications of FORTH can be seen in Chapter 7 below, where it is used as the basis of a games design package.

3

LOGO

Introduction

LOGO was first designed at Bolt, Beronek and Bewan Inc. in Cambridge, Massachusetts. It was written as part of an experiment to see if programming a computer could play a useful part in the education of children.

The earliest versions of LOGO, written by Wallace Fevrzieg, Daniel Bobrow and Seymour Papert, were very similar to another language called LISP, and dealt with the manipulation of words and sentences. It was soon discovered, however, that words and symbols did not stimulate the children in the way that had been hoped. The language continued to develop, the major change being the introduction of 'turtle graphics' by Seymour Papert. It is this aspect of LOGO which is most well known.

Because of the importance accorded to the graphical aspects of LOGO, over the years many versions have apeared that are not full implementations. This cannot be said of the Spectrum LOGO. As we will see, this implementation can be used to process words and sentences and to perform mathematical calculations, as well as to undertake the graphics applications that one would expect.

In many ways LOGO is similar to FORTH, with the computer performing the various actions as and when they are entered by the user. LOGO does not contain a very large vocabulary, just a few words which are often termed primitive procedures (or primitives for short). These primitives allow you to program the Spectrum in many different ways, and by using the procedure facility they can be used to develop more complex procedures which in turn allow yet more complex procedures to be defined.

LOGO is a very flexible language which can be used successfully by young children and experienced programmers alike. Because of its immediate mode of operation the user gets an instant reply to what he enters, and LOGO is therefore best learnt by experimentation and trial and error, methods which would not be applicable to the more conventional languages such as BASIC and Pascal.

The remainder of this chapter will be divided into two parts. The first will consider the graphical aspects of LOGO, dealing with simple movement of the turtle and the definition of more complex graphical procedures. The second part will consider text manipulation, an equally important aspect of the language, and its applications in the development of useful and amusing programs.

1. Turtle Graphics

When the machine is first switched on, it is in a form of text mode with the new cursor (?) in the top left-hand corner of the screen. To make the turtle appear we simply enter

```
? SHOWTURTLE
```

The screen will now be divided into two sections. The bottom two lines represent the area through which the user and the computer communicate and the rest of the screen is the field over which the turtle can be moved. When entering simple directional commands, the two lines at the bottom represent adequate space. If, however, we need to enter more complex routines which require more space, we can revert to a text mode by typing:

```
? TEXTSCREEN (TS)
```

which provides the normal 22 lines for communication.

Movement

There are four basic primitives which will cause the turtle to move and a line to appear on the screen:

FORWARD	FD
RIGHT	RT
LEFT	LT
BACK	BK

Each of these can be used in its full or abbreviated form and must be accompanied by one parameter. In the case of 'FORWARD' or 'BACK', this parameter represents the distance (number of pixels) to be moved; in the case of 'RIGHT' and 'LEFT', the parameter

indicates the amount of rotation in degrees. By using a combination of these simple steps it is possible to construct shapes on the screen.

Example 1

The following program can be used to produce a rectangle on the screen:

```
FORWARD  50
RIGHT    90
FORWARD  100
RIGHT    90
FORWARD  50
RIGHT    90
FORWARD  100
```

When this set of lines has been entered the rectangle will be complete but the overall presentation may be spoilt by the triangular turtle in the bottom left-hand corner. It can be removed by using the command:

HIDETURTLE (HT)

and then replaced by using the opposite command:

SHOWTURTLE (ST)

If we now wish to produce a second rectangle using a different colour scheme we must first clear the screen using:

CLEARSCREEN (CS)

and then define our new colours.

Colours are selected by using the three commands:

PENCOLOUR (PC)
BACKGROUND (BG)
SETBORDER (SETBR)

The PENCOLOUR command is used to change the INK colour, the BACKGROUND changes the paper and SETBORDER the border. Each of these commands has one associated parameter which must be an integer in the range 0 to 7 (the normal Spectrum colours).

Example 2

The following will produce a red square on a black background with a green border (enough to make any user ill).

```
SETBORDER   4
PENCOLOUR   2
BACKGROUND  0
FORWARD     50
RIGHT       90
FORWARD     50
RIGHT       90
FORWARD     50
RIGHT       90
FORWARD     50
```

There are two further commands which relate to the movement of the turtle and could be useful in the construction of pictures:

PENUP (PU)
PENDOWN (PD)

These two primitives are used to alter the state of the pen. If the pen is up, movement of the turtle will not cause a line to be drawn, while if the pen is down, any movement of the turtle will result in a line being drawn.

Other primitives which can be used with turtle graphics are listed and defined below.

CLEAN

This erases the graphics screen but does not alter the current position of the turtle.

DOT [x y]

This leaves a DOT in the current pencolour of the point defined by the co-ordinates (x,y).

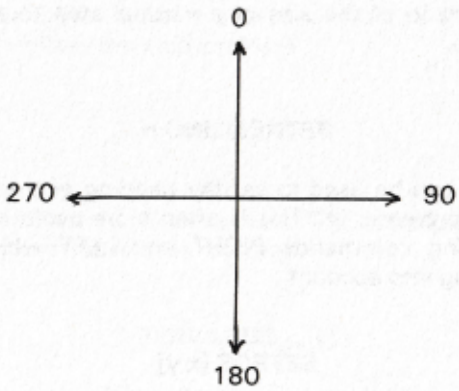
FENCE

This limits the movement of the turtle to the screen boundaries.

After FENCE has been defined any command which would result in the turtle going outside the limits will produce an error message.

HEADING

This returns a number in the range 0 to 359, showing the direction of the turtle. The normal compass system is used, as shown below:



HOME

Moves the turtle back to the centre of the screen and sets the heading to zero. If the pen is down then a line from the current position to the origin will be constructed.

PENERASE

A similar command to PENDOWN except that the turtle will now erase any lines which it passes over.

PENREVERSE

This puts the pen into a reverse mode, i.e. as it passes a pixel a line is erased if it exists and drawn if it does not.

POSITION

This returns the current position of the turtle.

SCRUNCH

A very useful operation which returns the current 'ASPECT RATIO' [x y], i.e. the ratio of the size of a vertical step to the size of a horizontal step.

SETHEADING n

This primitive can be used to set the heading of the turtle to the value of the parameter 'n'. This is often more useful than the two relative heading commands RIGHT and LEFT which take the current heading into account.

SETPOS [x y]

This moves the turtle to the position defined by the co-ordinates (x,y). Like the HOME command, a line is drawn if the pen is in the down position.

SETSCRUNCH [x y]

This powerful command can be used to set the scales of the horizontal and vertical motion of the turtle. The default values are [100 100] which means that if we move 100 units vertically or horizontally from the origin then we will reach the boundary of the screen. By using SETSCRUNCH the scale factors can be changed, and by making adjustments to x y, the overall shape of circles and squares can be changed. For example,

```
SETSCRUNCH (50 100)
```

SETX n

Moves the turtle so that the new x-co-ordinate is 'n' while leaving the y-co-ordinate unchanged.

SETY n

Similar to the SETX command except that the y-co-ordinate is changed rather than the x-co-ordinate.

SHOWNP

A simple operator which will return a value dependent on the state of the turtle. If the turtle is in the SHOWTURTLE mode it will return true, otherwise false is returned.

TOWARDS [x y]

An operator which returns the heading necessary to get from the current turtle position to the point specified by the co-ordinates (x y).

WINDOW

This command enables the turtle to move outside the screen area, even though the new position will not be displayed on the screen.

WRAP

This is the default condition of LOGO when the turtle graphics are first used. If the turtle crosses the screen boundary then it will automatically appear on the other side of the screen. This effect can be cancelled by using the FENCE or WINDOW command.

XCOR

Returns the x-co-ordinate of the current position of the turtle.

YCOR

Returns the y-co-ordinate of the current position of the turtle.

Now that we have defined all the primitives available in LOGO for the manipulation of turtle graphics, we can begin to look at the various programming methods that are available.

Repetition

One of the most important facilities in any language is the ability to repeat a particular statement or group of statements a set number of times. In BASIC this facility is made available by the use of FOR-NEXT loops. The equivalent in LOGO is REPEAT. The major difference is that LOGO is an immediate language and therefore the whole of the REPEAT statement has to be entered in one line.

The syntax of the REPEAT statement is as follows:

```
REPEAT n [LOGO PRIMITIVES]
```

The value of 'n' is the number of times that the collection of LOGO primitives within the square brackets will be executed.

Example 3

By using the REPEAT statement the square produced in Example 2 can be created more quickly by using only one line of program.

```
REPEAT 4 [FD 50 RT 90]
```

We see from this example that by using the REPEAT function it is possible to produce very short sections of code which will produce quite complex patterns. By nesting our REPEAT statements we can produce even more compact code.

Example 4

The following section of code uses two nested REPEAT statements in order to produce a complex pattern made from a series of squares.

```
REPEAT 20 [REPEAT 4 [FD 50 RT 90] RT 18]
```

Variables

All programming languages make use of variables. Unlike Pascal, which is a compiled language, the variables used in a LOGO program do not have to be declared, nor does their type have to be defined.

Variables are assigned in LOGO by using the MAKE primitive, as shown below:

```
MAKE "2 24
MAKE "LENGTH.1 36
MAKE "NAME "FRED
MAKE "FORWARD 22
```

Several important points arise from these examples:

1. Quotation marks are used to indicate that what follows is to be taken as a variable name.

2. Unlike BASIC and many other languages, the characters used in a variable name are not restricted to letters and digits only. In LOGO any character other than the right-square bracket (]), the left-square bracket ([) and the space can be used. This makes the following all acceptable as variables:

```
AREA(1)
WIDTH.3—RIGHT
FR+—/** (not very useful, but you never know).
```

3. There is no distinction between real, integer and character variables, and the same variable name could be used to contain a number or a string of characters:

```
MAKE "A 27
MAKE "A "FRED
```

Further methods of assigning values to variables will be considered in part 2 of this chapter.

4. When all variables are defined they are preceded by an identification character ("), and so it is possible to use a variable with the same name as a primitive such as FORWARD.

As primitives and variables can have the same name, confusion could arise if statements such as

```
FORWARD FORWARD
```

are used, since the computer would not have any indication as to which was the primitive and which the variable. This problem is easily overcome by using another identifier (a colon) to indicate the use of a variable. Therefore the correct version of the above line would be:

```
FORWARD :FORWARD
```

The colon indicates that the second occurrence of the word FORWARD is a variable and is therefore not considered as a primitive.

LOGO, like the majority of highly structured languages, makes use of both local and global variables. Any variable created as described here, by using the MAKE primitive, will be global. Any variable created for the specific use of a procedure (see below) and defined as part of the title of that procedure, will only be local. Once the procedure has been completed the values will be lost.

A programmer often does not know the values that will be assigned to a group of variables when the program is run. On such occasions, if programming in BASIC, he would use INPUT-type statements, which halt the execution of a program until the user enters the appropriate values. In LOGO this can be achieved by using the READCHAR or READLIST commands, which are defined as follows:

READCHAR (RC)

This waits for the user to press a key. The operation returns the character, which can then be checked or assigned to a variable. For example,

MAKE "A READCHAR

This is very similar to the INKEY statement used in BASIC, as it only allows one key to be pressed, and the character is not automatically printed on the screen.

READLIST (RL)

This returns whatever is typed at the keyboard until the 'ENTER' key is pressed. The list can then be checked or assigned to a variable. For example,

MAKE "B READLIST

This is the LOGO equivalent of INPUT, with each character being printed in the text area of the screen as it is entered.

Example 5

The following program uses the READLIST primitive together with a single variable to produce a square, with the length specified by the user.

MAKE "LENGTH READLIST REPEAT 4 [FD : LENGTH RT 90]

When this has been entered the computer will pause until a value has been entered for READLIST and assigned to the variable length. The correct size of square will then be produced on the screen.

Procedures

We have seen that LOGO is an immediate language, with the lines of code being executed as soon as they are entered. This provides the programmer with an electronic calculator or drawing-board type of environment, but it restricts the length of programs that can be entered to a single line of code, which can contain any of the pre-defined primitives which constitute LOGO.

From a graphical viewpoint the primitives available are very simple and several have to be used in order to construct even the most simple of shapes. For example to construct a rectangle we would require:

```
FORWARD 50
RIGHT   90
FORWARD 80
RIGHT   90
FORWARD 50
RIGHT   90
FORWARD 80
```

This can be shortened by using the repeat statement

REPEAT 2 [FORWARD 50 RIGHT 90 FORWARD 80 RIGHT 90]

but this still seems an excessive amount of code to produce a simple shape which we may require on numerous occasions.

It soon becomes obvious that it would be very useful if we had a primitive called RECTANGLE which could be used in the same way as the other primitives such as FORWARD or RIGHT. Although no such primitive exists, LOGO provides the programmer with a powerful command 'TO' which enables new primitives, generally known as procedures, to be written. Consider the problem of defining a procedure called 'RECTANGLE' which will produce a rectangle (80 × 50) whenever it is called. This can be achieved by entering:

```
TO RECTANGLE
FORWARD 50
RIGHT   90
FORWARD 80
RIGHT   90
FORWARD 50
RIGHT   90
FORWARD 80
END
```

When defining a procedure there are some important points to note.

1. When the first line of the procedure, known as the title line,

TO RECTANGLE

has been entered, the computer will enter 'TO MODE'. A new prompt (>), instead of (?) will appear and the procedure can then be entered one line at a time without the computer executing each instruction as is normally the case.

2. Every procedure must finish with the line END. It tells LOGO that you have finished the definition. The procedure will then be translated into code but not executed, a message such as

RECTANGLE defined

will be displayed, and the direct 'LOGO MODE' will be re-entered with the normal cursor (?) re-appearing on the screen.

The editor

Using 'TO' to define a procedure has two drawbacks. First, if the screen is currently in a graphics mode then there are only two lines available at the bottom of the screen, and it is often very difficult to define a procedure without being able to reference the previously entered lines. Secondly, when using the 'TO MODE' to enter a procedure we do not have any editing facilities, and if a mistake is made it cannot be corrected.

An alternative method of defining a procedure is to enter the 'EDIT MODE' by typing EDIT followed by the name of the procedure to be defined. For example,

EDIT RECTANGLE

If there is a procedure called RECTANGLE, which has already been defined, then the screen will be cleared and the complete definition will be displayed. If no definition exists then the appropriate title line

TO RECTANGLE

will appear at the top of the screen, and the procedure can then be entered in the usual manner. Using the editor is a much more convenient way to define procedures, as errors can be corrected and alterations made simply by moving the cursor around the screen.

The following table represents the facilities available when using the editor built in to the Spectrum LOGO.

CAPS 5	Moves cursor one character left
CAPS 6	Moves cursor one character down
CAPS 7	Moves cursor one character up
CAPS 8	Moves cursor one character right

CAPS 0	Deletes one character
EMODE CAPS 5	Moves cursor to beginning of line
EMODE CAPS 6	Moves cursor to end of screen
EMODE CAPS 7	Moves cursor to beginning of screen
EMODE CAPS 8	Moves cursor to end of line
EMODE B	Moves cursor to beginning of text
EMODE E	Moves cursor to end of text
EMODE Y	Deletes line from position of cursor and saves
EMODE R	Enters the 'saved line' at cursor position
SYS S	Stops scrolling
EMODE P	Moves cursor to previous page
EMODE N	Moves cursor to next page
EMODE C	Exits editor, incorporating all modifications into procedure definition

Several of the editor commands refer to 'EMODE'. This can be entered by pressing 'CAPS SHIFT' and 'SYMBOL SHIFT' simultaneously until the letter 'E' appears in the bottom left-hand corner of the screen. To exit from E MODE, repeat the process until the letter 'C' appears. Anyone familiar with the normal Spectrum editor used in BASIC will appreciate the power and simplicity of this system.

When a procedure has been defined by using TO or EDIT, it can be used in the same way as any of the primitives supplied with the LOGO package. To draw a rectangle on the screen we simply move the turtle to the required position and type RECTANGLE, and a rectangle (80 × 50) will appear on the screen.

A simple procedure such as the one to construct a rectangle has one major drawback: it is only capable of producing a rectangle of a pre-defined size. It is possible, however, to define procedures which can accept inputs as and when they are called.

Consider the problem of defining a procedure which will take two inputs (parameters) called length and width and construct a rectangle of an appropriate size. This can be achieved as follows:

```
TO RECTANGLE :LENGTH :WIDTH
FORWARD :WIDTH
RIGHT 90
FORWARD :LENGTH
RIGHT 90
FORWARD :WIDTH
RIGHT 90
FORWARD :LENGTH
END
```

In this particular case the variables LENGTH and WIDTH are examples of local variables, and any values assigned to them in the procedure call will only last until the procedure has been completed, at which time their values will return to any global values which may have been previously defined.

This new RECTANGLE procedure can now be used at any time simply by typing its name, followed by two numbers. For example,

```
RECTANGLE 120 40
RECTANGLE 30 70
RECTANGLE :LEN :WID
```

In the third of these examples we have used two variables LEN and WID. The values of these are assigned to LENGTH and WIDTH respectively and the procedure is executed.

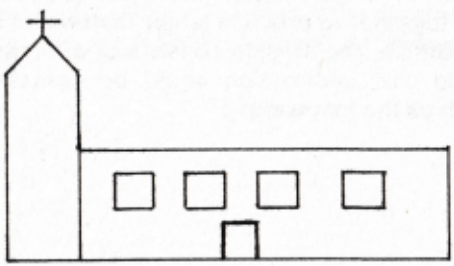
Principles of LOGO programming

Writing a complex program in LOGO generally consists of defining a number of procedures, each of which will contain either the pre-defined primitives or yet other procedures which must then be defined. The final program therefore may consist of one procedure which uses two other procedures which in turn make use of even more procedures and so on.

> So, naturalists observe, a flea
> Hath smaller fleas that on him prey;
> And these have smaller fleas to bite 'em,
> And so proceed ad infinitum
>
> Jonathan Swift

Example 6

Consider the problem of designing a church like the one shown below. Careful inspection shows that the church consists of rectangles for the main building, tower, door and windows, a triangle for the steeple, and a cross. It is therefore necessary to define each of these shapes as separate procedures, and then combine them to produce the final drawing.



The first stage is to create the procedures for drawing the three basic shapes, rectangle, triangle and cross.

```
TO RECTANGLE :BASE :HEIGHT
FORWARD :HEIGHT
RIGHT 90
FORWARD :BASE
RIGHT 90
FORWARD :HEIGHT
RIGHT 90
FORWARD :BASE
RIGHT 90
END

TO TRIANGLE :BASE
RIGHT 30
FORWARD :BASE
RIGHT 120
FORWARD :BASE
RIGHT 120
FORWARD :BASE
RIGHT 90
END


TO CROSS :VERT :HORIZ
FORWARD :VERT
RIGHT 180
FORWARD :VERT/3
RIGHT 90
FORWARD :HORIZ/2
RIGHT 180
FORWARD :HORIZ
RIGHT 180
FORWARD :HORIZ/2
LEFT 90
FORWARD :VERT*2/3
RIGHT 180
END
```

Now that the three basic shapes have been defined, we can link some of them together to produce larger features of the complete church. For example, the steeple consists of a rectangle, triangle and cross, and this information could be contained within a procedure such as the following:

```
TO STEEPLE :BASE :HEIGHT
RECTANGLE :BASE :HEIGHT
PENUP
FORWARD :HEIGHT
PENDOWN
TRIANGLE :BASE
PENUP
RIGHT 30
FORWARD :BASE
LEFT 30
PENDOWN
CROSS :BASE :BASE/2
END
```

Similarly the block of four windows could be created using a single procedure, making use of the rectangle procedure and the REPEAT primitive:

```
TO WINDOWS :SIZE
REPEAT 4 [RECTANGLE :SIZE :SIZE
PENUP
RIGHT 90
FORWARD :LENGTH/4.5
LEFT 90
PENDOWN]
END
```

Now that the individual procedures have been completed, we can combine them to create the church. However, before we do this we will construct one final simple procedure which will move the turtle back to some constant position from which all references will be taken.

```
TO RESET
PENUP
SETPOS [-128 -88]
PENDOWN
END
```

This simply moves the turtle to the bottom left-hand corner of the screen, where we are going to construct the diagram. We are now in a position to create the final procedure, as follows:

```
TO CHURCH :LENGTH :HEIGHT
RESET
STEEPLE :LENGTH/3 :HEIGHT*1.5
RESET
RIGHT 90
PENUP
FORWARD :LENGTH/3
LEFT 90
PENDOWN
RECTANGLE :LENGTH :HEIGHT
RIGHT 90
FORWARD :LENGTH/2
LEFT 90
RECTANGLE :LENGTH/5 :HEIGHT/4
RESET
RIGHT 90
PENUP
FORWARD :LENGTH*4/9
LEFT 90
FORWARD :HEIGHT *2/3
PENDOWN
WINDOWS :LENGTH/9
HIDETURTLE
END
```

To construct a church on the screen is now a simple matter. We just type:

```
CHURCH 100 60
```

where 100 and 60 are the two parameters used for the main building, and all the other measurements are based on these. By using this system, the drawing will always be in correct proportion no matter what values are used for the parameters.

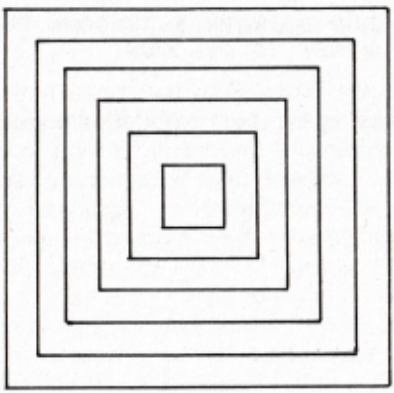
This method is generally known as 'bottom up programming', as we start by defining very simple procedures which are used as part of the definition for more complex procedures, etc.

The alternative approach, known as 'top down programming', is to start by defining the main procedure, i.e. 'CHURCH', and then define the smaller procedures that are required by it.

This method will generally lead to more compact code, but it is more complicated to write and is generally used only by more experienced programmers.

Recursion

According to Seymour Papert, one of the designers of LOGO, recursion is the most valuable aspect of LOGO and indeed of computer languages in general. Recursion is the name given to a specific case when the definition of a procedure partly consists of itself. Consider the problem of producing a tunnel consisting of a number of squares, each slightly smaller than the previous one, as shown below.



This could be produced by defining a procedure as follows:

```
TO TUNNEL :SIZE
REPEAT 4 [FORWARD :SIZE RIGHT 90]
PENUP
RIGHT 45
FORWARD 2
PENDOWN
LEFT 45
TUNNEL :SIZE -4
END
```

You can see from this that our definition of the procedure 'TUNNEL' contains a direct call to itself. This is possible because when the call is made the procedure has already been defined. It is an excellent example of recursion and demonstrates how small recursive routines can be used to generate quite complex pictures.

Animation

One of the most useful aspects of recursion is the production of animation, a topic which is very difficult to master. It is beyond the scope of this book to consider animation in detail, but we will describe the general approach by considering a simple problem.

Most churches have a clock on the tower, but the one that we have produced does not. We will now remedy this situation by defining two routines, one of which is recursive, in order to produce a clock with a moving hand.

The face of the clock does not require any animation and can be constructed by using the following simple procedure:

```
TO CLOCK
REPEAT 12 [PENUP FORWARD 5 PENDOWN FORWARD 2
PENUP BACK 7 RIGHT 30 PENDOWN]
END
```

To construct a moving hand we can define a recurring procedure as follows:

```
TO HAND
HIDETURTLE
PENDOWN
FORWARD 5
PENUP
BACK 5
WAIT 50
ERASE
FORWARD 5
PENUP
BACK 5
RIGHT 30
HAND
END
```

The complete picture of a church with an animated clock can be produced by adding:

```
RESET
PU
FD :HEIGHT
RT 90
FD :LENGTH/6
LT 90
PD
CLOCK
HAND
```

```
HT
END
```

to the end of the 'CHURCH' procedure and then entering

```
CHURCH 100 60
```

as before.

In this first part of the chapter we have scratched the surface of the turtle graphics side of LOGO. LOGO is a language which is learnt by experiment rather than formal teaching methods, and we hope that you have built up an appetite to learn more about this powerful language.

2. Manipulation of Lists

In cannot be overstressed that LOGO does not simply consist of turtle graphics. A quick glimpse through any manual will indicate that the number of primitives relating to graphics is only a small portion of the total number available. However, any article or book written on the language is usually heavily biased towards graphics. The powerful commands for handling words and lists are often under-documented.

The rest of this chapter will consider this aspect of the language. The meaning of the terms 'word' and 'list' will be defined, as will the procedures available for their manipulation.

In many ways LOGO is similar to the artificial intelligence language LISP, and they share many common features. In LISP the basic unit is known as the 'atom', and every line of program or section of data is divided up into a number of these atoms. In LOGO, however, the basic unit for data is known as a 'word', and any item of data will consist of a number of words joined together.

The rules governing the structure of a word were considered earlier in this chapter when we investigated the use of variables within a program. The most important point to remember is that a space cannot be part of a word as it acts as a delimiter in the construction of lists. This does not pose much of a limitation as each of the following could represent a legal word.

```
HELLO!
FIRST.LENGTH
123
```

The third example above is the most interesting, as using a

number as a word or variable would be totally impossible in BASIC or many other computer languages. It can, however, lead to problems and the programmer must therefore take care. For example,

```
MAKE "2 4
PRINT "2          will print 2.
PRINT :2          will print 4 (the value of "2).
```

Having defined the LOGO word we can now define a list, which is simply a collection of words that are to be considered together. Each word is separated by a comma from the next, and the whole list must be enclosed within square brackets.

The following represent legal lists in LOGO:

```
[THIS IS A LIST]
[A LIST CONTAINING A NUMBER 24]
[A LIST CONTAINING [A LIST]]
```

The third demonstrates the recursive nature of LOGO, in that we are able to define a list, which contains a list, which contains yet another list, etc.

Related primitives

Here we give a resumé of the primitives available in Spectrum LOGO for forming and manipulating words and lists.

ASCII

This returns the ASCII code of the given character. For example,

```
PRINT ASCII "A ——› 65
PRINT ASCII "* ——› 42
```

BUTFIRST

BUTFIRST can be applied to a word or a list and returns everything but the first element. For example,

```
PRINT BUTFIRST "WORD ——› ORD
PRINT BUTFIRST "[THIS IS A LIST] ——› IS A LIST
```

CHAR

This is the opposite of the ASCII primitive and returns a character, given its code. For example,

```
PRINT CHAR "65 ——› A
```

The quotation marks are very important, because if we type

```
PRINT CHAR 65
```

the computer would take 65 to be a procedure, which would not have been defined, and so an error message would be produced.

COUNT

This counts the number of elements in a word or list. For example,

```
PRINT COUNT "WORD ——› 4
PRINT COUNT '[THIS IS A LIST] ——› 4
PRINT COUNT '[THIS IS A [DOUBLE LIST]] ——› 4
```

EMPTYP

This returns TRUE if the LOGO object is empty, otherwise FALSE. For example,

```
PRINT EMPTYP "[ ]" ——› TRUE
```

EQUALP

This takes two parameters which can be words or lists and compares them, returning TRUE if they are identical and FALSE if not. For example,

```
PRINT EQUALP "HELP "HELP ——› TRUE
PRINT EQUALP "[A LIST] "[A LIST] ——› TRUE
PRINT EQUALP "[HELLO] "HELLO ——› FALSE
```

FIRST

A useful primitive which returns the first element of a word or list. Those readers familiar with LISP will see that this is the same as the CAR operation. For example,

```
PRINT FIRST "HELLO ——› H
PRINT FIRST "[A LIST OF WORDS] ——› A
```

FPUT

This takes two parameters: an object (word or number) and a list, and places the object at the start of the list. For example,

```
FPUT "HELLO "[THIS IS ME]
```

will produce the new list [HELLO THIS IS ME]

ITEM

A useful primitive which extracts an item from a list. For example,

```
PRINT ITEM 4 [THIS IS A COMPLEX LIST] ——› COMPLEX
```

LAST

This returns the last element of a word or list. For example,

```
PRINT LAST [THIS IS A LIST] ——› LIST
PRINT LAST "WORD ——› D
```

LIST

This is often referred to as a 'greedy primitive' as it can take any number of parameters, which it then forms into a list. For example,

```
MAKE "TEST LIST "CAT "DOG "MOUSE
PRINT :TEST
```

will produce the output [CAT DOG MOUSE]

LISTP

This returns TRUE if the object is a list, otherwise FALSE. For example,

```
PRINT LISTP [A SMALL LIST] ---> TRUE
PRINT LISTP "WORD ---> FALSE
```

LPUT

This is similar to FPUT except that the object is appended to the end of the list. For example,

```
LPUT "LIST [THIS IS A SMALL]
```

will produce the new list [THIS IS A SMALL LIST]

MAKE

This is used to assign values to variables. The values themselves can be numbers, words or lists. For example,

```
MAKE "A1 426
MAKE "A2 "FRED
MAKE "A3 [A LIST OF SMALL WORDS]
```

When a variable has been assigned in this way it can be used by prefixing its name with a colon (:)

MEMBERP

This returns TRUE if the object is an element of the list, otherwise FALSE. For example,

```
PRINT MEMBERP "TEST [THIS IS A TEST] ---> TRUE
```

NUMBERP

This returns TRUE if the object is a number, otherwise FALSE. For example,

```
PRINT NUMBERP 3 ---> TRUE
PRINT NUMBERP "4 ---> FALSE
PRINT NUMBERP [7] ---> FALSE
```

NAMEP

A useful primitive which can be used to test whether a value has been assigned to a variable. It returns TRUE if the object has a value, otherwise FALSE.

```
PRINT "NAMEP "FRUIT ---> FALSE
MAKE "FRUIT "APPLE
PRINT NAMEP "FRUIT ---> TRUE
```

SENTENCE

This is a greedy operation which will take any number of parameters and form them into a list.

Although it apears to be the same as LIST, SENTENCE has an extra facility in that it strips the bracket from any of the inputs that is a list. For example,

```
PRINT LIST "A "SMALL [LIST OF WORDS] ---> [A SMALL [LIST OF WORDS]]
PRINT SENTENCE "A "SMALL [ LIST OF WORDS] ---> [A SMALL LIST OF WORDS]
```

WORD

This joins the various objects together to form a single word. For example,

PRINT WORD "HOT "DOG —→ HOTDOG

WORDP

This returns TRUE if the object is a word, otherwise FALSE. For example,

PRINT WORDP "B64 —→ TRUE
PRINT WORDP [A] —→ FALSE

RUN

This takes a LOGO list, which is then executed as if the list were a line of program. For example,

RUN [PRINT [HELLO]] —→ HELLO

PRINT

The PRINT command is probably the most important in any language, as without it no output can ever be displayed on the screen. In LOGO the print statement can take one object which is produced on the screen. This object can be a word, list, numerical calculation or the value of some previously defined variable. For example,

PRINT "HELLO —→ HELLO
PRINT [HELLO READER] —→ HELLO READER
PRINT "4 + 6 —→ 10
PRINT :FRED —→ (the value of FRED)

Where next?

Having read through the last few pages, you may well be of the opinion that there is a large number of primitives available for the manipulation of data in the form of words and lists. However, when you start programming it will soon become obvious that the primitives available are very limited and unlikely to provide the facilities that you require. Similar problems are encountered with turtle graphics, in that commands to draw simple shapes

(rectangle, triangle, etc.) do not exist as part of the LOGO system, and therefore have to be defined as procedures.

Working with textual data is not very different from using turtle graphics as we approach any problem in the same way, i.e. we use the general primitives in order to define more complex procedures which in turn form part of yet more complicated procedures.

Consider the problem of defining a procedure 'REPLACE' which would take three inputs, two words and a list and then replace every occurrence of the first word by the second word throughout the list. The main problem in this case is that there is no primitive for extracting the nth element in a list and we must therefore use FIRST and BUTFIRST in a recursive way to step through the list.

```
TO REPLACE :OLD :NEW :LI
IF :LI = [] THEN OUTPUT []
IF EQUALP :OLD (FIRST :LI) OUTPUT FPUT :NEW
REPLACE :OLD :NEW (BUTFIRST :LI)
IF NOT EQUALP :OLD (FIRST :LI) OUTPUT FPUT FIRST :I
REPLACE :OLD :NEW :LI
END
```

In this recursive routine the OUTPUT command is used to exit the loop with its argument being returned to the calling procedure. When you have entered this procedure it can be tested as follows:

```
PRINT REPLACE "GOOD "BAD [THE GOOD AND THE UGLY]
```

By defining such procedures we can create a new LOGO working environment from which we can expand by defining more and more sophisticated routines.

Conclusion

In this chapter we have considered many LOGO commands and operations relating to two aspects of the language: turtle graphics and list manipulation. This does not represent the sum total of possible applications of LOGO, however; a full understanding of its form and usefulness can only be gained by personal experimentation.

4

Micro-PROLOG

Introduction

Ever since the first programmable computers were developed, computer languages have been based on the processing of numbers, or 'number-crunching' as it is affectionately known. All the well-known languages, for example BASIC, COBOL and FORTRAN etc, are essentially based on handling numbers. Indeed until now this has been the only real task of all computers.

However, we are standing on the threshold of a new computer age. As the speed of operation of computers increases and they become able to store much more information, we are moving towards machines that will be able to handle knowledge and concepts rather than binary numbers. Computers will be able to approach problems in a similar way to the human brain, making decisions by trial and error and learning from their mistakes by adding new pieces of data to expand their 'brain size'. The computer may select certain logical paths to follow, missing out others in order to save time. A new generation of computer languages is being developed for this new generation of computers, and one of these languages is Micro-PROLOG.

The main difference between Micro-PROLOG and languages like BASIC is essentially the way in which they operate. A BASIC program can most simply be described as an ordered list of instructions that are performed to achieve an end result. For example, here is a simple BASIC program:

```
10 LET A = 6
20 LET B = 3
30 LET C = A/B
40 IF C = INT (C) THEN PRINT A;" IS A MULTIPLE OF
50 STOP
```

This program will decide whether A is a multiple of B. If it is, an appropriate message will be displayed on the screen. The computer performs this task by working through the instructions, executing each one as it is reached. It is quite easy to follow this

example, since high-level languages are 'descriptive' and this quality is the basis of the language Micro-PROLOG.

A program written in Micro-PROLOG is a list of logical statements and inter-relationships between them, which form a type of database. The execution of the program involves posing a query about information in the database in order to discover some relationship, instead of the normal execution of a set of instructions, which happens in other high-level languages.

How Micro-PROLOG works

Here are two statements for a Micro-PROLOG program that we will be developing throughout this chapter.

```
Wild-Boys by Duran-Duran
Hungry-Daze by Deep-Purple
```

Consider the first statement. We have two pieces of data linked by the relationship, 'by'. This is known as a binary relationship. Spaces are important, and so with a piece of data or a relationship there can be no spaces; hence the use of the hyphen.

If we wish to place these two statements into the program (database), we must use the instruction 'add'.

There should be a prompt on the screen ('&.'), after which two lines may be typed, with any extra being ignored:

```
&. add (Wild-Boys by Duran-Duran)
&. add (Hungry-Daze by Deep-Purple)
```

Remember that the prompt is supplied and need not be entered. Now type 'list all' and you will see both the statements displayed. These two statements have been added to the computer's database, but at present this is all that it knows.

We can query the database by asking questions about it. For instance, if we wish to verify that Wild-Boys is by Duran-Duran we would ask 'is Wild-Boys by Duran-Duran?', and the Micro-PROLOG form of this is exactly that:

```
is (Wild-Boys by Duran-Duran)
```

The reply will be 'YES'. If, instead, you had typed:

```
is (Wild-Boys by Deep-Purple)
```

You would have got the reply 'NO'.

To extend the database, add the following statements in the usual way.

```
Jump by Van-Halen
The-Riddle by Nick-Kershaw
Relax by Frankie-Goes-To-Hollywood
New-Song by Howard-Jones
What-is-Love by Howard-Jones
We-Rock by Dio
Evil-Eyes by Dio
Locomotion by OMD
American-Heartbeat by Survivor
Two-Tribes by Frankie-Goes-To-Hollywood
Faithfully by Journey
Van-Halen group
Deep-Purple group
Survivor group
OMD group
Dio group
Frankie-Goes-To-Hollywood group
Nick-Kershaw soloist
Howard-Jones soloist
Duran-Duran group
```

Now type 'list all'', and all the statements will be listed.

You should have noticed that the last few statements have a different structure to the rest. They are known as unary statements and they describe a piece of data. For example,

```
Duran-Duran group
```

defines Duran Duran as a group. The 'is' instruction can be used with these statements in the same way as described for binary statements. For example,

```
is (Deep-Purple group)
```

will get the reply 'YES'.

As you can probably imagine, there are many more instructions to query the database than just the 'is' statement. For instance, you may wish to enquire of the database, 'Which tracks were by Howard-Jones?'. This is achieved by issuing the following statement:

```
which (x : x by Howard-Jones)
```

The part of the instruction after the colon is known as the 'query condition' and determines how the database is tested. The letter before the colon is known as the 'answer pattern' and defines which part of the query condition is printed.

The computer searches through the database, comparing each statement to the query condition, remembering that 'x' can stand for anything, and if a match is found, the answer pattern is displayed. In this case the answer pattern is whatever 'x' happens to be. The search does not stop with the first match, it carries on until the end of the database, displaying all the matches it locates.

Try typing in the query given above. The following will be displayed:

```
New-Song
What-Is-Love
No (more) answers
```

The same instruction can be applied to unary statements. For instance, if you wish to find the names of all the solo artists stored in the database, you could use the following query:

```
which (x : x soloist)
```

You will receive the answer:

```
Nick-Kershaw
Howard-Jones
No (more) answers
```

You may want to find the titles of all the records by soloists, the actual names of the soloists being unimportant. In this case you would have to use a conjunctive query, i.e. a question with more than one condition. The command for this is:

```
which (x : x by y and y soloist)
```

The computer will search through the database using the first conditon 'x by y' until it finds a match. When it does, it switches control to the second condition, 'y soloist', and then searches through the database using this condition. If a further match is found, x will be displayed and control will return to the first condition and move on to the next statement in the database,

otherwise control passes back with no effect.

The actual content of y is unimportant, as it can be thought of as a 'dummy' variable, only being used during the actual search. It could, however, be included in the answer pattern by extending the instruction to:

```
which (x y : x by y & y soloist)
```

An ampersand can be used instead of the word 'and'.

Text can also be displayed in the answer pattern if required, but care must be taken on entering proper spaces. When this command is issued:

```
which (x BY y : x by y & y soloist)
```

the following will be displayed:

```
The-Riddle BY Nick-Kershaw
New-Song BY Howard-Jones
What-is-Love BY Howard-Jones
No (more) answers
```

The same approach to conjunctive queries may be applied to the 'is' questions. You may want to know whether 'Relax' is by a group or a soloist; in such an instance, you could use the following query:

```
is (Relax by y & y group)
```

to which you would receive the reply 'YES' as Frankie Goes to Hollywood is a group.

Rules

More often than not, conjunctive queries will be asked of the database and typing them in becomes longwinded. It would therefore be much easier if we could define a relation based upon a conjunctive query. Also we may wish to draw conclusions from the relations defined in the database. For instance, the fact that Wild-Boys is by Duran-Duran obviously means that Duran-Duran performed 'Wild-Boys', but to the computer this is unclear. Therefore we must have some way of defining new relations using those already defined.

In order to make this section clearer some more data should be entered into the database.

D-L-Roth vocalist-for Van-Halen
E-Van-Halen guitarist-for Van-Halen
A-Van-Halen drummer-for Van-Halen
M-Anthony bassist-for Van-Halen
I-Gillan vocalist-for Deep-Purple
R-Blackmore guitarist-for Deep-Purple
R-Glover bassist-for Deep-Purple
J-Lord keyboards-for Deep-Purple
I-Paice drummer-for Deep-Purple
S-Le-Bon vocalist-for Duran-Duran
A-Taylor guitarist-for Duran-Duran
R-Taylor drummer-for Duran-Duran
J-Taylor bassist-for Duran-Duran
N-Rhodes keyboards-for Duran-Duran
S-Perry vocalist-for Journey
S-Smith drummer-for Journey
N-Schon guitarist-for Journey
J-Caine keyboards-for Journey
R-Valory bassist-for Journey

Not all the names of the members of the groups have been included, and it is left to you to extend the database further if you wish.

Now type 'list all' and you will see a list of all the statements so far entered into the database, grouped into specific sets. You can list just one set of statements if you wish, by typing 'list' followed by the relation relating to that set. For example, 'list guitarist-for' would list all the statements containing the relation 'guitarist-for'.

To find out which drummer 'y' played on a certain track 'x', you would use the following query:

which (y DRUMMED-ON x : x by z & y drummer-for z)

The computer will search through the database using the first condition 'x by z' until it finds a match; then it will switch control to the second condition 'y drummer-for z' and search through the database trying to find a second match. It already knows the contents of z, being the name of a group (or soloist) given to it by the first condition. So it then tries to find the name of the drummer (y) in the group (z). If it can find one, it will display the answer pattern. Control will then be switched back to the first

condition, and it moves on to the next statement in the database.

Using this query, we can create the rule 'drummed-on', and this acts in the same way as a relation. The rule is typed into the computer in the following way:

```
add (y drummed-on x if x by z & y drummer-for z)
```

If you now type 'list all', you will see the rule listed at the end of the statements.

This rule is a convenient way of creating a new set of statements using the relation 'drummed-on', without the need to type all the new statements into the database. This new set would be:

```
A-Van-Halen drummed-on Jump
I-Plaice drummed-on Meanstreak
S-Smith drummed-on Faithfully
etc.
```

The same method can be applied to vocalists, guitarists, etc. Using this method, it is also possible to create the rules for:

1. sung-on
2. played-guitar-on
3. played-bass-on
4. played-keyboards-on

Now, by using the following query, all the vocalists and the tracks they sang on can be displayed:

```
which (x SUNG-ON y : x sung-on y)
```

If we only wanted one answer to be displayed at a time, the query 'one' would be used in the following way:

```
one (x SUNG-ON y : x sung-on y)
```

After each answer is displayed, the computer asks if any more are required and you will need to reply 'y' or 'n' to the prompt.

Now let us extend the database by adding some album titles. Enter the following statements in the usual way.

```
1984 album-by Van-Halen
Perfect-Strangers album-by Deep-Purple
```

Frontiers album-by Journey
Arena album-by Duran-Duran

For convenience, all the tracks already in the database apply to the above groups and are on the above albums.

Defining rules from rules

It is also possible to create new rules using rules that have already been defined. For example, we could define a rule 'on-album' using the 'album-by' and 'by' relations. Enter the following rule:

```
x on-album y if y album-by z and x by z
```

This rule can be tested by using various queries, such as:

```
is (Wild-Boys on-album Arena)
all (x : x on-album Frontiers)
```

The first query will return 'YES' if 'Wild Boys' is on 'Arena'; the second will list all tracks known to the database that appear on the album 'Frontiers'.

Now, using this rule and the 'drummed-on' rule, we can create a new rule 'drummed-on-album'. It could take the following form:

```
x drummed-on-album y if
z on-album y and
x drummed-on z
```

This new rule will work perfectly if the database stays as it is, but a problem will arise if there is more than one track per album in the database. If the following statement is entered:

```
Meanstreak by Deep-Purple
```

and we query the database to list all occasions on which the 'drummed-on-album' rule is satisfied using the following:

```
which (x y : x drummed-on-album y)
```

the result should be two listings for:

```
I-Plaice Perfect-Strangers
```

This is simply because the 'dummy' variable 'z' is filled and match is found twice as there are two tracks. Therefore it is important to think through the construction of rules carefully before implementing them. The following rule is a better version and will solve this problem:

```
x drummed-on-album y if
y album-by z and
x drummer-for z
```

Using this method it is possible to create the following rules:

1. sung-on-album
2. played-bass-on-album
3. played-guitar-on-album
4. played-keyboards-on-album

We will now consider a new topic in Micro-PROLOG, so it would be a good idea to save the program as it stands using the 'SAVE <filename>' command. Now use the 'KILL ALL' command to delete the program from the computer's memory.

Lists

So far this chapter has only considered statements that concern individual items. For example, if we wished to express the titles of four tracks that comprise an album, we would use the following:

```
1984 on-album 1984
Jump on-album 1984
Panama on-album 1984
Top-Jimmy on-album 1984
```

However, these four statements can be entered using the single statement:

```
(1984 Jump Panama Top-Jimmy) on-album 1984
```

The titles of the tracks are put together in a structure called a list. After the statement has been entered into the database, we can query it in the following way:

```
which (x : x on-album 1984)
```

This will produce the reply:

```
(1984 Jump Panama Top-Jimmy)
No (more) answers
```

In order to get at an individual element within the list, we have to expand on the ideas of pattern-matching that we have encountered so far in this chapter.

Fixed-length lists

First we need to introduce some more statements into the database:

```
(Holy-Diver Gypsy Invisible) on-album Holy-Diver
(Open-Fire Forever Black-Tiger) on-album Black-Tiger
(New-Song Equality Natural) on-album Humans-Lib
(Faithfully Back-Talk Frontiers) on-album Frontiers
```

These statements list only three tracks recorded on each album, and so they have the following form:

```
(x1 x2 x3) on-album y
```

By using the query.

```
all (y : (x1 x2 x3) on-album y)
```

We can find the names of all the albums containing three tracks. In the same way, we can find the names of all the albums containing four tracks:

```
all (y : (x1 x2 x3 x4) on-album y)
```

In this case, we would only receive one answer: '1984'.

There is no limit to the number of lists that can be used within a statement. If required, it is possible to replace the title of the album by a list containing two elements: the title of the album and the name of the group (or soloist).

We can illustrate this point by deleting the relation 'on-album' using the command 'KILL on-album' and entering the following statements:

(Holy-Diver Gypsy Invisible) on-album (Holy-Diver Dio)
(Open-Fire Forever Black-Tiger) on-album (Black-Tiger Y-and-T)
(New-Song Equality Natural) on-album (Humans-Lib Howard-Jones)
(Faithfully Back-Talk Frontiers) on-album (Frontiers-Journey)

The statements now take the following form:

(x1 x2 x3) on-album (y z)

Since this 'pattern' matches every statement, we can use it to define rules. For example,

x on-album-by z if x on-album (y z)

If this is entered then a query of the form:

which (x : x on-album-by Dio)

can be used to locate the names of the tracks on any album by Dio stored in the database and will return:

(Holy-Diver Gypsy Invisible)
No (more) answers

Lists do not have to contain only individual elements, they can also contain other lists. So it is possible for each statement to take the following form:

(x1 x2 x3) on-album ((x y) z)

Included within the second list is a third list. In this case, x may represent the title of the album and y the side number, with z being the artist. A typical statement would look like:

(Rock-Rock Photograph Stagefright) on-album
((Pyromania 1) Def-Leppard)

Using this, you could enquire of the database which side of an album the above three tracks are on:

which (y : (Rock-Rock Photograph Stagefright)
on-album ((x y) z)

Notice that 'x' and 'z' have been used in place of 'Pyromania' and 'Def-Leppard', as we are not concerned with what these actually are.

Variable length lists

In the last section we looked at fixed length lists. This worked because we knew the exact pattern of every statement. However, in the real world not all albums have the same number of tracks on them, so a query such as:

```
which (x1 x2 : (x1 x2 x3) on-album (y z))
```

will not be feasible as not all the statements will follow this pattern. For example, the rules:

```
x1 by z if (x1 x2 x3) on-album (y z)
x2 by z if (x1 x2 x3) on-album (y z)
x3 by z if (x1 x2 x3) on-album (y z)
```

work correctly if each album has only three tracks, but if there are albums in the database with any number of tracks other than three, then these rules will not recognise them. This problem is overcome by using a rule such as:

```
x by z if Y on-album (y z) and z contained-in Y
```

where Y is any list of tracks on album y. To use this the relation, 'z contained-in Y' must be defined. If we take a list of, say, six elements:

```
(x1 x2 x3 x4 x5 x6)
```

it can be divided into two parts – the element 'x1' followed by the list '(x2 x3 x4 x5 x6)'. In Micro-PROLOG, this is known as the pattern (x | y). 'x' is the first element of a list followed by 'y', the rest of the list. The pattern can be demonstrated by entering the following query:

```
which (x y : (x|y) on-album (Humans-Lib Howard-Jones))
```

Since we already know that the complete list defined in the statement is:

(New-Song Equality Natural)

'x' will contain the element New-Song and 'y' the list (Equality Natural).

The pattern (x | y) is not restricted to just one form. Another example of its use:

which (x y z : (x y | z) on-album (Humans-Lib Howard-Jones))

This time, the pattern returns two elements 'x' and 'y', followed by the list 'z', and the above query would return:

New-Song Equality (Natural)
No (more) answers

Returning to the original problem of defining the rule 'contained-in', we can say:

1. x is contained in a list if it is the first element; or
2. x is contained in a list if it is in the list following the first element.

The first statement can be written simply as:

x contained-in (x | z)

If the above statement is executed and the two 'x's do not match, then it must be true that x is not the first element but contained within 'z'. The next statement is then executed:

x contained-in (y | z) if x contained-in z

In effect what happens is that the first element 'y' is removed from the list, 'z' being the new list. The first statement is executed again and the first element of the new list is checked. The effect of the two statements is to search through the list until a match is found between the x and the first element in the list. This is known as a recursive relation.

Now enter the two rules into the database and we can query the database about certain elements of lists. For example,

is (Y on-album (1984 Van-Halen) and Jump contained-in Y)

In the above query, Y is defined as the list containing the tracks on

the album '1984', i.e. (1984 Jump Panama Top-Jimmy). The query 'Jump contained-in Y' is then posed.

When the first 'contained-in' rule is met, '1984' is the first element in the list and therefore the rule is not satisfied. The second rule is then encountered and the list 'z', (Jump Panama Top-Jimmy), is passed back to the first rule. The second time, 'Jump' is the first element in the list and therfore the rule is satisfied and the query confirmed. 'YES' appears on the screen.

Now try this query:

is (Y on-album (1984 Van-Halen) and Breathless contained-in Y)

This time the list Y becomes shorter and shorter until it becomes an empty list '()'. In this case a match cannot be found and the query cannot be confirmed. 'NO' appears on the screen.

The rules can also be used to give a list (in the literal sense) of all the tracks in the database as follows:

all (x : y on-album (z1 z2) and x contained-in y)

See if you can work out how it operates yourself.

Arithmetic

Micro-PROLOG is designed to handle language, facts, etc., but often you may need to do some arithmetic as well. Micro-PROLOG contains four pre-defined arithmetic relations. It is worth noting that each relation acts as though it were a query for some large database.

1. SUM

The 'SUM' relation can be used for addition, subtraction and verification, and it takes the following form:

SUM (x y z)

where x+y=z

(a) Addition

which (x : SUM (4 5 x))

This query will return '9', since 4+5=x: x=9.

(b) Subtraction

which (x : SUM (4 x 9))

This query will return '5', since 4+x=9: x=9–4=5.

(c) Verifying

is (SUM (4 5 9))

This query will return 'YES'.

Note that there must be only one unknown.

2. TIMES

The TIMES relation can be used for multiplication, division and verification. It takes the following form:

TIMES (x y z)

where x * y = z. We will use '*' to represent multiplication.

(a) Multiplication

which (x : TIMES (4 5 x))

This query will return '20', since 4*5=x : x=20.

(b) Division

which (x : TIMES (4 x 20))

This query will return '5', since 4*x=20 : x=20/4=5.

(c) Verifying

is (TIMES (4 5 20))

This query will return 'YES'.

As with the 'SUM' relation, there must be only one unknown.

3. INT

The INT relation has two uses. It can either be used to see if a number is an integer, or it can be used to convert a floating-point number into an integer.

(a) Testing to see if a number is an integer.

```
is (15 INT)
```

This query will return 'YES'.

```
is (4.36 INT)
```

This query will return 'NO'.

(b) Converting a floating point number into an integer.

```
which (x : 4.36 INT x)
```

This query will return '4'.

4. LESS

This query is used to check that one number is less than another. It takes the following form:

```
(x LESS y)
```

where x is less than y. It is used as follows:

```
is (5 LESS 7)
```

This query will return 'YES'.

```
is (8 LESS 2.5)
```

This query will return 'NO'.

The LESS relation can also be used with letters to check the alphabetical order. For example,

```
is (RICHARD LESS STEPHEN)
```

This query will return 'YES'.

Like the other relations, the arithmetic relations can be used in conjunctive queries. For example,

```
which (y : TIMES (3 x 9) & SUM (x 4 y))
```

will return 7 since 3*3=9 ——> x=3 and 3+4=7 ——> y=7.

Rules can also be defined using arithmetic relations. For example, the relation 'factor-of-12' can be defined by:

```
x factor-of-12 if TIMES (x y 12) AND y INT
```

The rule works by dividing 12 by 'x'; if the remainder is an integer, then 'x' must be a factor of 12.

Conclusion

Obviously there is a lot more to Micro-PROLOG than has been revealed in this chapter, but we hope that we have provided enough information here to whet your appetite for further study.

5

Pascal

Introduction

Second only to BASIC, Pascal is one of the most widely used languages on modern microcomputers. It was invented by Nicklaus Wirth in the late 1960s and designed to teach structured programming techniques to students. However, it quickly developed into much larger spheres of application than simply education.

Pascal has the advantage of being a highly portable language since, with very few exceptions, it has a standard set of rules that are common to all versions. This coherence gives Pascal a considerable advantage over a language like BASIC which seems to have an almost infinite number of dialects.

The other major benefit of Pascal is the speed at which programs can be executed, since the language is compiled into machine code and this code is then executed directly. Although it is possible to write badly structured programs in Pascal, we will see that even the slowest Pascal solution to a given problem is considerably faster than the most efficient solution in BASIC.

As an example of this, consider the following two programs to determine whether a given number is prime (i.e. divisible only by 1 and itself), the first written in BASIC and the second in Pascal:

```
10 CLS
20 INPUT "NUMBER = ";N
30 FOR I = 2 TO SQR(N)
40 IF INT (N/I) = N/I THEN PRINT N;" IS NOT PRIME : END
50 NEXT I
60 PRINT N; "IS PRIME"
70 END
```

```
10   PROGRAM PRIME;
20   VAR
30     I,N:INTEGER;
40     FLAG:BOOLEAN;
50   BEGIN
60     FLAG = TRUE;
```

```
 70     I:= Z;
 80     READLN(N);
 90     WHILE (I<SQRT(N)) AND (FLAG = TRUE) DO
100       BEGIN
110         IF TRUNC(N/I) = N/I THEN FLAG:= FALSE;
120         I:= I+1
130       END
140     IF FLAG = TRUE THEN WRITELN (N,' IS PRIME'
        ELSE WRITELN (N,' IS NOT PRIME')
150   END.
```

By trying both of these individual programs for large prime numbers, such as 29789, you can see how efficient Pascal is, since both programs employ the same algorithm. It should be mentioned here that although the line numbers in BASIC form an integral part of the code, their only purpose in Pascal is to provide a convenient reference system when a program is to be edited.

The language

All programs in Pascal contain three vital ingredients;

1. A program name, otherwise known as an identifier.
2. A declaration section in which all variables (and constants) to be used during the program are declared. This section also includes all the functions and procedures to be used in the main program.
3. The main program.

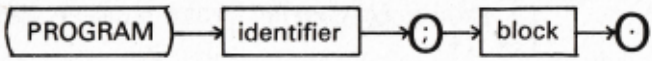
These three sections are illustrated in the Pascal program given above. The program name 'PRIME' is given in line 10, the variables are declared in lines 20 to 40, and the main program is from line 50 onwards.

PROGRAM and syntax diagrams

A program is given a name by using the reserved word 'PROGRAM' and the main program is contained within the two reserved words 'BEGIN' and 'END.'. The full stop after END is important since it signifies the end of the program rather than the end of some other operation, which also uses the word END.

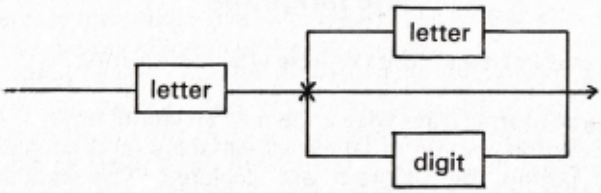
Because of its structured nature, anything in Pascal can be

represented pictorially by using a syntax diagram. The first syntax diagram represents the whole program:



Syntax diagrams show how legal statements can be constructed in Pascal and hence produce a program that is syntactically correct. In the diagrams anything enclosed in a circle or in a box with circular ends represents a reserved word or part of syntax which must be entered exactly as written. All other rectangles represent either a Pascal concept which will require further definition in another syntax diagram, or something which requires no further comment such as a letter. If in your Pascal statement it is possible to follow the flow of the lines connecting the boxes from the left side to the right then the statement will be correct.

The syntax diagram for an identifier is as follows:



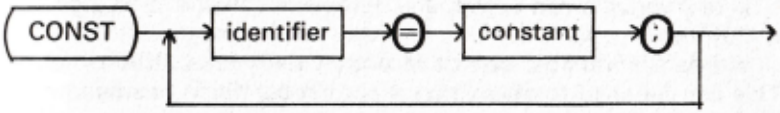
Reading this diagram from the left, the box 'letter' signifies that all program names must start with such a character. It is then possible to loop around the 'letter' path or the 'digit' path at will until an exit is made via the right end arrow. Consequently, B, BAT and BO4N are all valid identifiers, but 24N is not.

Since this book is not designed to be an exhaustive manual, we will not reproduce all the syntax diagrams required to define Pascal. A complete set of the diagrams can be found in most books dedicated to Pascal.

VAR and CONST

Perhaps the most important part of a Pascal program is the declaration section, because it is here that you must tell the computer the names of all variables you are going to use and specify the type of values they will take. It is also possible to

define constants which will not vary throughout the program, by giving them a name and stating their value. The constant declaration section starts with the reserved word CONST and has a syntax diagram as follows:

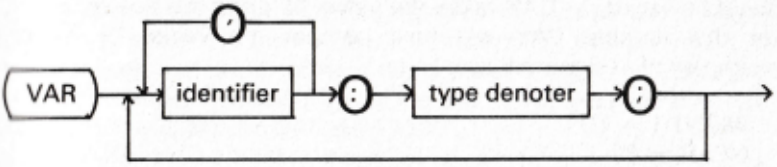


We will not develop the diagram any further, but rather illustrate the possible range of values of CONST by a few examples:

```
CONST
  DAYSINWEEK = 7;
  HOURSINDAY = 24;
  FIRSTDAY = 'SUNDAY';
  PI = 3.141592654;
  E = 2.718281828;
```

By now you will no doubt have noticed quite a large number of semi-colons in the syntax diagrams and examples, appearing after some but not all statements. These are extremely important, and their use is well defined. The purpose of a semi-colon is to separate statements, and attention must be paid to where they are to be placed since a failure to comply with the syntax structure will result in error 2 – 'semi-colon expected' – appearing when you try to compile your program.

As mentioned above, all variables to be used throughout the program must be declared together with information on the type of values they will take. The syntax diagram for this is:



It can be seen from the diagram that the variable declaration is more complicated than that for the constant and that it starts with the reserved word VAR. There are many different possible contents for the 'type denoter' box, including some structures

which the user can create himself, but for the time being we will consider the three basic types: integer, real and Boolean.

An integer variable is a whole number in the range −32768 to 32767, and the language also supports the reserved constant MAXINT, set to 32767.

A real variable can accept any numerical value in the range -3×10^{38} to $+3 \times 10^{38}$.

A Boolean variable can be in one of two states: true or false. This can be used to check that a certain condition has been met before exiting from a loop.

An example of variable declaration is thus:

```
VAR
   COUNT, DAY, DATE, MONTH : INTEGER
   RADIUS, AREA : REAL
   ATWORK, ATHOME : BOOLEAN
```

Notice the use of a comma to separate variables of the same type; this is to overcome the need to have a separate line for each variable.

Assignment

We can regard variables, once they have been declared, as empty boxes waiting to be filled with some value. To give a variable a value there is a special symbol known as the assignment operator ':=' which should be read as 'takes the value'. So the statement:

```
DAY := 15
```

should be read as 'DAY takes the value 15'; and the box reserved for the variable DAY will now contain the value 15. Other examples of assignment are:

```
MONTH := 10
YEAR := 67
COUNT := COUNT + 1
ATHOME := TRUE
ATWORK : = FALSE
RADIUS := 10
AREA := PI*RADIUS*RADIUS
```

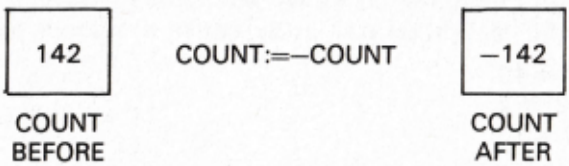
The use of an assignment operator as opposed to an equality sign does make it easier to understand a statement like COUNT:=COUNT+1. It should be read as 'COUNT takes the value of COUNT plus one'; this is easier to understand than COUNT = COUNT+1, which looks like a mathematical equation that can be re-arranged to show that 0=1!

In the last example the value of AREA is determined by the value of RADIUS (another variable) and the value of PI (a constant defined above). The * represents multiplication.

This use of pre-defined constants or variables can be extended to allow such lines as:

```
COUNT := -COUNT
```

where the new value of COUNT is equivalent to minus one multiplied by the old value (COUNT := (−1)* COUNT). For example,



Arithmetic and integers

Given any two numbers, it is possible to obtain a third by performing an arithmetical calculation such as addition, subtraction, multiplication or division. Addition is denoted by '+', subtraction by '−' and multiplication by '*' as in BASIC, but when it comes to division we must be more careful, since dividing one integer by another does not always produce an integral result. If we are using real numbers then the usual operator '/' will produce the real number quotient, but if we are using integers then we must use the operators DIV and MOD.

DIV will produce the result of a division, neglecting any fractional part which may occur, and MOD will return the remainder. Thus

```
14 DIV 4 = 3
14 MOD 4 = 2
14/4 = 3.5
```

since 14/4 is 3 with remainder 2 or 3.5.

Pascal observes the normal priorities for arithmetic functions:

1. Predeclared functions.
2. Multiplication and division.
3. Addition and subtraction.

Parentheses can be used to override these rules, any operation contained within brackets being performed first.

In Pascal there is no operator for raising one number to the power of the other as there is in BASIC (↑), so if such a facility is required it must be written by the programmer. A simple function to do this will be written later in this chapter, but for now let us consider the two pre-declared functions which take integers for their arguments and return integral results.

The first is SQR(x), which calculates the square of a number. (Programmers familiar with BASIC will have seen a SQR(x) function before, but in BASIC the function finds the square root.) The second is ABS(x) which returns the absolute value of x. For example,

SQR(7) = 49
ABS(−2) = 2
ABS(3) = 3

Arithmetic and real numbers

Real numbers can be expressed in the conventional notation (e.g. 137.45) or standard index notation, which is convenient for very large or very small numbers. In the standard index notation the letter 'E' represents 'ten to the power of'. For example,

$2E8 = 2 \times 10^8 = 200000000.0$
$5E{-}4 = 5 \times 10^{-4} = 0.0005$
$-1.3745E2 = -1.3745 \times 10^2 = -137.45$

It must be remembered that when the computer is using real numbers it can only store a certain number of digits. Consequently, if 'infinitely long decimals' are created, such as 1/3 = 0.3˙ (0.3 recurring), the calculations will be subject to rounding error. A demonstration of this can be obtained with an ordinary calculator: if you enter the value 2, find its square root ten times and then square this value ten times, it is most unusual to obtain

the answer 2.

The two predeclared functions SQR(x) and ABS(x) can also be used with real numbers as well as integer numbers, along with the following, which will produce a real number as the result:

SIN(x)	returns the sine of x
COS(x)	returns the cosine of x
ARCTAN(x)	returns the arctangent of x
EXP(x)	returns the value of e raised to the power of x
LN(x)	returns the natural logarithm of x
SQRT(x)	returns the square root of x

The arguments for the first two functions and the result of the third are angles expressed in radians (360 degrees = 2×PI radians).

Finally, there are two predeclared functions which take real arguments and produce integer results. These are ROUND(x) and TRUNC(x). The first of these, ROUND(x), will round the number to the nearest integer; the second, TRUNC(x), will return the integer part of the number x in the same way as INT in BASIC. For example,

ROUND (23.5) = 24
TRUNC(23.7) = 23

Input/output

Armed with the above information we are now in a position to write a program in Pascal. As an example, here is a program to calculate the area of a circle with radius 5.

```
10 PROGRAM CIRCLE;
20 CONST
30   PI=3.141592654;
40   RADIUS=5;
50 VAR
60   AREA:REAL;
70 BEGIN
80   AREA:=PI*SQR(RADIUS);
90 END.
```

The ';' in line 80 is not strictly necessary but makes the following additions easier.

This program can be entered into the Spectrum, the code can

be compiled into machine instructions, and it can be executed – all without error. It will not, however, tell us the result because there is no instruction telling the computer to give any output. The Pascal equivalents of PRINT in BASIC are the functions WRITE and WRITELN. Thus we can instruct the program to tell us the values of RADIUS and AREA by adding the following lines:

```
85 WRITELN (RADIUS);
86 WRITELN (AREA)
```

This will produce the output:

```
5
7.85398E+01
```

Remember to recompile every time the program is amended, otherwise the computer will execute the last compiled version of the program.

Well, this is a start, but it is not the most useful of output formats, and there are many ways in which it can be improved. To start with, it would be nice if the output described what it was displaying, i.e. radius and area. This can be achieved by amending lines 85 and 86 to

```
85 WRITELN ('RADIUS = ',RADIUS);
86 WRITELN ('AREA = ',AREA)
```

Anything in a WRITE or WRITELN instruction that is enclosed within single quotes will be regarded as text and will appear on the screen exactly as in the program, so the output is now

```
RADIUS = 5
AREA = 7.85398E+01
```

The value of the integer radius can be displayed in a print field of any width. This means that we can right justify values very easily since the values are displayed with enough spaces at the start of the field to allow the number to fit exactly. For example,

```
85 WRITELN (RADIUS = ', RADIUS :7);
```

will result in an output of

```
RADIUS =       5
```

where six spaces have been added in front of the digit 5 to make the print field seven characters wide.

A similar technique can be employed to display a real number, such as the area, in a more readable format. This is done by specifying the field width and the number of digits to appear after the decimal point. So if the value of PI were to be displayed on the screen, lines like the following would produce the shown output:

```
WRITELN (PI:12:6)     3.141593
                     ----------
                     12 characters

WRITELN (PI:10:2)       3.14
                     ----------
                     10 characters
```

It is important to remember that the decimal point occupies one of the character positions in the field.

Therefore to make the final alterations to the example program the lines 85 and 86 should be amended as follows:

```
85 WRITELN ('RADIUS = ',RADIUS);
86 WRITELN ('AREA = ',AREA:6:2)
```

which will produce the output

```
RADIUS = 5
AREA = 78.54
```

We have mentioned two statements for producing output on the screen, WRITELN and WRITE, but up to now we have only examined the first. The WRITE statement is exactly the same as the WRITELN, except that it does not select a new line after it has displayed its contents. Hence

```
85 WRITE ('RADIUS = ',RADIUS);
86 WRITELN (' AREA =' ,AREA:6:2)
```

will produce the output

```
RADIUS = 5 AREA = 78.54
```

This is perfectly adequate if the radius of the circle we are concerned with is five, but if we wish to select a different value we

do not want to have to re-write the program, re-compile and re-execute. If you are familiar with BASIC, you will know that it is useful to give values to variables during the execution of a program and that to do this the INPUT statement is used. To generate the same effect in Pascal we use the READ statement, and to demonstrate its use, consider again the program CIRCLE on p. 117 above.

If line 40 is removed and line 60 is converted to

```
60 RADIUS, AREA: REAL;
```

The computer will now expect the RADIUS to be a variable and hence its value can be specified during program execution. This is achieved by adding the line

```
75 READ(RADIUS);
```

so that the program can now be listed as

```
10 PROGRAM CIRCLE;
20 CONST
30   PI = 3.141592654;
50 VAR
60   RADIUS,AREA:REAL;
70 BEGIN
75   READ(RADIUS);
80   AREA:= PI*SQR(RADIUS);
85   WRITELN('RADIUS =',RADIUS:6:2);
86   WRITELN('AREA   =',AREA:6:2)
90 END.
```

Line 85 has been slightly amended to tidy up the output format of the radius since the variable RADIUS is now of type real and no longer an integer.

When this program is executed (after it has been compiled) it will appear that nothing is happening since all you will see on the screen is a flashing C. This is the cursor, and it is prompting you to type in a value for the radius. Try typing the number 5 followed by the 'ENTER' key and you will see on the screen:

```
5
RADIUS =  5.00
AREA   = 78.54
```

Notice the way that the output is right justified by the effect of the format command in the WRITELN statement.

Although a flashing C informs you that the computer is waiting for a value to be input, it is not very clear; if you were unaware of the functions of the program you would not know what the computer was asking for. This can be remedied by displaying a short message before the prompt with a WRITE statement:

```
74 WRITE('ENTER RADIUS ');
```

Now when the program is executed the message 'ENTER RADIUS' appears in front of the flashing cursor: If we again enter 5 the output will appear as:

```
ENTER RADIUS 5
RADIUS = 5.00
AREA = 78.54
```

It should be noted that if the value of the area will not fit into the field description the output automatically reverts to standard index form. Try a radius of 30, and you will get the following screen display:

```
ENTER RADIUS 30
RADIUS = 30.00
AREA = 2.82743E+03
```

since the area is 2827.43 and the number cannot be expressed in the format 6:2, which only allows three digits before the decimal point.

There is also a READLN command and this has the same effect as a READ statement except that it will select the next line of input after reading an element of data. To demonstrate this, suppose that A and B are two variables in a program and that at some point we encounter the commands:

```
READ(A);
READ(B);
```

then an input of 5 7 will result in A=5, B=7. However, should the statements be

```
READLN(A);
READ(B);
```

then an input of 5 7 will result in A=5, but the 7 will be ignored and the value of B will be requested on the next line.

Comments

It is often advisable to insert comments into a program to describe the action of the following lines, so that on reference to a listing at a later date the operation of each section of the program is immediately apparent. In BASIC this is done by using a REM statement and this slows the program down because each REM is interpreted before it is ignored. In Pascal, comments can be placed anywhere in a program (except in the middle of a symbol) and are contained between the special brackets '(*' and '*)'. When the program is compiled, all spaces and comments are completely ignored and consequently have no influence on the machine code which is to be executed. So an addition such as

```
5(*FIND THE AREA OF A CIRCLE *)
```

will have no effect on program execution but serves as a reminder of the object of the program. It is not necessary to place a separating semi-colon after a comment.

Boolean variables

Boolean algebra, named after George Boole (1815-1864), is closely related to logic and deals with statements that are either true or false. TRUE and FALSE are constant identifiers in Pascal, denoting the only two values that a Boolean variable can take.

The most common way of obtaining a Boolean value is by comparing two numbers of suitable type. For example, 5>3, 9<12 are TRUE and 4<=3, −7>−2 are FALSE. As can be seen from these examples, it may be necessary to combine some of the symbols (<= represents the mathematical sign ⩽), and it should be remembered that keys for this purpose exist on the Spectrum keyboard.

Armed with these comparisons it is possible to force the computer to follow a certain path through a program or make it repeat a section of code until a condition is true, and these methods of controlling program flow will be considered later in this chapter. It must be stressed that a comparison such as A>B is not a statement of fact, but a test on a certain condition that can

be used to control flow at a particular point in a program.

It is frequently necessary to test for more than one condition at any one time, so Pascal supports the Boolean operations AND, OR and NOT to combine comparisons. The operators AND and OR both take two Boolean operands and return a Boolean result, and their operation can best be shown by considering a 'truth table' which lists all possible outcomes:

x	y	x AND y
F	F	F
F	T	F
T	F	F
T	T	T

x	y	x OR y
F	F	F
F	T	T
T	F	T
T	T	T

AND returns a TRUE response if both operands are TRUE, and OR returns TRUE if either or both of its operands are TRUE.

The third Boolean operation is NOT, which takes a single Boolean value as its operand and yields a Boolean result which is the opposite of the operand.

x	NOTx
F	T
T	F

Thus a combination of conditions can now be tested. For example,

(A<B) and (B<C)
A AND (NOT B)

Very often brackets are included to make the syntax easier to follow, but in the first example given above they are necessary because the comparisons (<, <=, >, >=, =, <>) are of the lowest priority and without the brackets the program would try to evaluate B AND B first, causing an error.

It is possible to have Boolean constants and variables, with the variables being assigned values using the assignment operator as follows (FLAG having been declared as a Boolean variable):

FLAG := TRUE

or

FLAG := (3>2)

or

FLAG := (A>B) and (B>C)

They can also have their value (TRUE or FALSE) displayed on the screen by using a WRITE statement. For example,

```
WRITELN(FLAG);
```

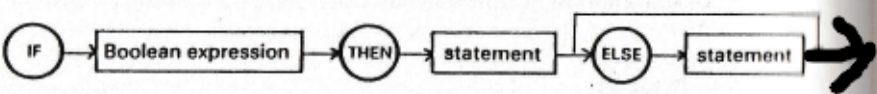
Control structures

Four different methods can be used to alter the flow of control of a program, each acting in a slightly different way.

1. IF ... THEN ... ELSE

IF, THEN and ELSE are all reserved words in Pascal. By using them a particular section of program can be selected and the remainder neglected. This acts in a different way to the IF ... THEN ... GOTO structure in BASIC, since all the program to be performed when the condition is satisfied comes with the statement in Pascal. This is similar to a BASIC statement which places all actions to be performed after the IF ... THEN, separating them by colons.

The syntax diagram for IF ... THEN ... ELSE is as follows:



The first statement is executed if the Boolean expression is TRUE and the second statement is executed if the expression is FALSE. It can be seen from the syntax diagram that the ELSE clause can be omitted, with the flow continuing to the next section of program. The following are legal IF statements in Pascal:

```
IF A=3 THEN
  WRITELN('THREE')
ELSE
  WRITELN('NOT THREE')
```

```
IF (A=3) AND (B=4) THEN
  WRITELN('A+B=7')
```

Note that there are no semi-colons in the actual IF statement itself; they only appear if the contents of the statement box require more than one operation. In the second example the ELSE clause has been omitted as it is not required.

Suppose, however, that you also wish to display the product and the difference of A and B if their values are 3 and 4 respectively. This could be done by using three IF ... THEN statements, but that would be clumsy and there is a much more elegant solution to the problem:

```
IF (A=3) AND (B=4) THEN
  BEGIN
    WRITELN('A+B=7');
    WRITELN('A*B=12');
    WRITELN('A-B=-1')
  END
```

By enclosing the instructions to be performed inside a BEGIN and END, it is possible to make the computer execute them all if the condition is true. Here is an example program to find the surface area or volume of a cube, depending on the answer to a simple question

```
 10 PROGRAM CHOICE;
 20 VAR
 30   DECISION : INTEGER;
 40   LENGTH,AREA,VOLUME : REAL;
 50 BEGIN
 60   WRITE('AREA(1) OR VOLUME(2)?');
 70   READ(DECISION);
 80   WRITE('LENGTH OF SIDE =');
 90   READ(LENGTH);
100   IF DECISION =1 THEN
110     BEGIN
120       AREA:= 6*LENGTH*LENGTH ;
130       WRITELN('AREA = ',AREA:8:3)
140     END
150   ELSE
160     BEGIN
170       VOLUME:=SQR(LENGTH)*LENGTH;
180       WRITELN('VOLUME = ',VOLUME:8:3)
190     END
200 END.
```

Note that a semi-colon is not required after a BEGIN statement and ·t the end of any line before an END statement. The program has been written with the two sections contained in the IF statement indented in an attempt to make the flow of the program easier to understand. The program is syntactically correct, but from a purist point of view it falls down since the program will calculate the volume not only if you select choice 2, but if this number is anything other than 1. This could be corrected in two ways. First, the following lines could be added:

```
75 IF (DECISION=1) OR (DECISION=2) THEN
76  BEGIN
195 END
```

This encompasses all the main program (lines 80 to 190) inside a BEGIN END pair and only allows this to be executed if DECISION holds the value 1 or 2. Alternatively you could add the following line:

```
155 IF DECISION =2 THEN
```

This will only allow the ELSE clause to be performed if DECISION holds the value 2. In this solution we have used an IF statement inside another IF statement, and this is particularly useful when certain conditions are to be tested. For example,

```
IF (A<>0) AND (B/A = 4) THEN
```

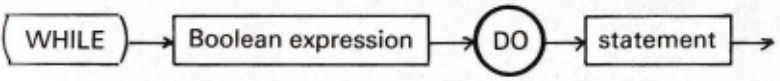
will cause an error if A=0 since the computer will try to evaluate B/A, i.e. divide by zero. The way around this is to use two IF statements, one inside the other. For example,

```
IF A<>0 THEN
   IF B/A=4 THEN
```

2. WHILE ... DO

Another fundamental control structure is the 'loop'. This enables a particular section of program to be repeated over and over again until a certain condition is reached. If the condition never occurs, the loop will continue indefinitely since there is no escape. When this occurs we call it an infinite or continuous loop.

The syntax diagram for a WHILE statement is as follows:



WHILE and DO are reserved words. The statement can be just one instruction or a set of instructions contained between a BEGIN/END pair. The sequence of operations is shown by the following algorithm:

1. Test the Boolean expression. If it is FALSE continue from step 4.
2. The condition must be TRUE, so execute the statement.
3. Repeat from step 1.
4. Since the expression is FALSE, continue with the remaining program.

Thus an example of a legal loop is the following, in which the computer counts from 1 to 10000:

```
10 PROGRAM COUNT;
20   VAR A:INTEGER;
30 BEGIN
40    A:=1;
50    WHILE A<>10000 DO A:=A+1
60 END.
```

The program continues to add 1 to the value of A until the condition A<>10000 becomes FALSE, i.e. when A=10000.

It is instructive to compare the time taken for this with the time it takes a computer to count from 1 to 10000 using BASIC. The Pascal program takes just over 1 second, while the BASIC version requires 1 minute 20 seconds!

In the above example there is only one instruction inside the loop, but we can include as many as we wish.

Here is a Pascal program that uses WHILE ... DO to find the factorial value of a number, N, followed by the corresponding program in BASIC (factorial N = N! = N x (N−1) x....x 1).

Pascal

```
10 PROGRAM FACTORIAL;
20   VAR N,FACT:INTEGER;
30 BEGIN
```

```
 40    WRITE('WHAT IS THE NUMBER');
 50    READLN(N);
 60    FACT:=1;
 70    WHILE N<>1 DO
 80      BEGIN
 90        FACT:=FACT*N ;
100        N:=N-1
110      END
120    WRITELN('FACTORIAL ',N,' = 'FACT)
130 END.
```

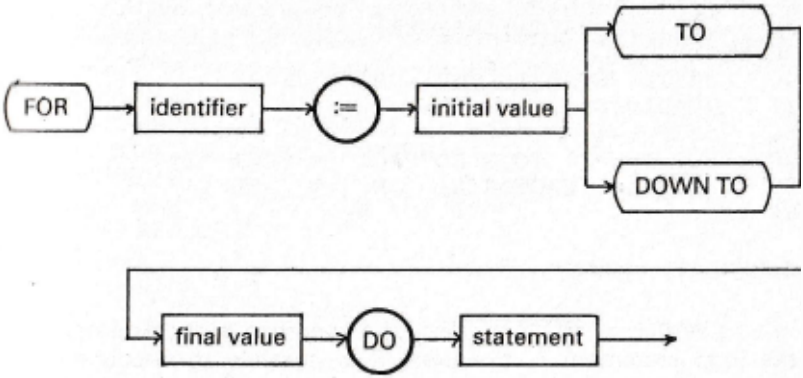
BASIC

```
10 CLS
20 INPUT"WHAT IS THE NUMBER";N
30 LET FACT = 1
40 IF N = 1 THEN GOTO 80
50 LET FACT = FACT*N
60 LET N =N-1
70 GOTO 40
80 PRINT"FACTORIAL ";N;" = ";FACT
90 END
```

Note that factorial numbers get large very quickly: 8! = 40320. Thus any integer larger than 7 will cause an overflow error. Moreover any integer less than 1 will cause an infinite loop and then an overflow error.

3. FOR ... DO

The FOR ... NEXT command in BASIC allows a particular section of code to be repeated a certain number of times, with the code to be executed contained between the FOR statement and the NEXT. In Pascal the FOR ... DO statement is slightly different, as can be seen from the following syntax diagram:



Again the statement can be either one instruction or a set of instructions contained within a BEGIN and an END. Also, there is no equivalent of the BASIC STEP facility, since the increment will always be one. FOR is a reserved word in Pascal. Providing the number of repetitions is known, a FOR ... NEXT loop can be employed. Examples of legal statements are

```
FOR I := 1 TO 10 DO WRITE('*');
FOR I := 10 DOWN TO 1 DO N:=N+I;
```

It is also possible to use variables to define the initial and final values, as long as they are of an appropriate size. For example,

```
FOR I := N DOWN TO 1 DO FACT := FACT*N;
```

or

```
FOR I:= A TO B DO
   BEGIN
      SUM = SUM+I;
      PRODUCT = PRODUCT*I
   END
```

It is clear from these examples that if the initial value is greater than the final value then the reserved words DOWN TO must be used, otherwise the single word TO is employed.

Using this control structure, the program to find N! can be written in a slightly more elegant way:

```
10 PROGRAM FACTORIAL2;
20   VAR I,N,FACT :INTEGER;
30 BEGIN
40   WRITE('WHAT IS THE NUMBER');
50   READLN(N);
60   FACT :=1;
70   FOR I := 1 TO N DO FACT := FACT *I;
80   WRITELN('FACTORIAL ',N,' = ',FACT)
90 END.
```

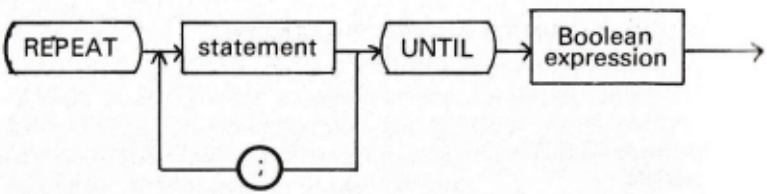
4. REPEAT ... UNTIL

With a WHILE ... DO loop, since the condition is checked before the loop statement is executed, it is possible to encounter a situation where the loop is not performed at all. For example, if as a reply to the prompt 'WHAT IS THE NUMBER', you type the value 1, when the program encounters the line

WHILE N <> 1 DO

it will ignore the loop code since N= 1.

Occasionally it is necessary to have a loop in which the first iteration is performed regardless of the condition which is checked after the execution of the loop code. This can be achieved by using the REPEAT ... UNTIL control structure. Its syntax diagram is as follows:



The effect of a REPEAT ... UNTIL loop is as follows:

1. Obey the statement or statements between REPEAT and UNTIL.
2. Test the value of the Boolean expression and if it is FALSE go back to step 1.
3. Since its value must be TRUE, continue with the program.

By comparing this with the algorithm for the WHILE ... DO loop

you can see that this guarantees the iteration to be performed once. Although the factorial program does not require such a facility, here is another version, this time using REPEAT ... UNTIL:

```
 10 PROGRAM FACTORIAL3;
 20   VAR N, FACT :INTEGER;
 30 BEGIN
 40   WRITE ('WHAT IS THE NUMBER');
 50   READLN(N);
 60   FACT := 1;
 70   REPEAT
 80     FACT := FACT*N;
 90     N:= N-1
100   UNTIL N = 0 ;
110   WRITELN ('FACTORIAL ',N,' = ', FACT)
120 END.
```

Functions

Often when writing programs a calculation has to be performed several times throughout the code and this can become very tedious and untidy. To overcome this repetition Pascal allows you to define your own functions, similar to COS, SIN, etc.

When a function is declared at the beginning of a program you must give the computer the following information:

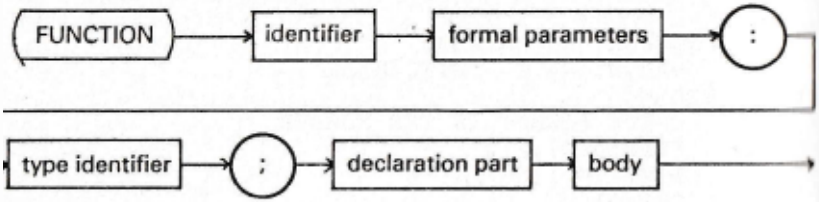
1. The function's name.
2. The parameters and their types.
3. The type of the result of the function.
4. The statements which compute the value.

Because we may wish to apply the functions to many different variables throughout the program, in the function definition we introduce a new variable which will only be used in the function. We call this a formal parameter.

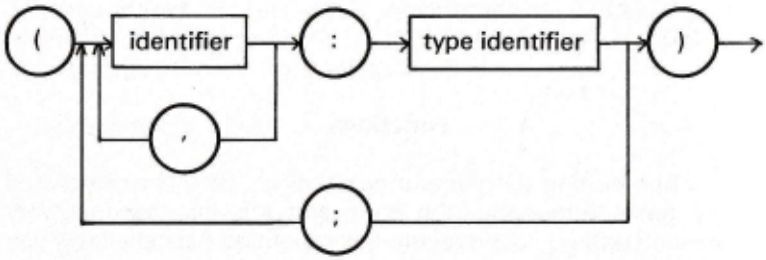
We have already mentioned that Pascal offers no convenient way of raising a number to a given power, so here we will develop a function to perform this task. Consider raising the number X to the power Y. In the function definition, we must specify

1. The function's name: POWER.
2. The parameters : X, Y are both REAL.
3. The type of the result : REAL.
4. The body of the function.

If we follow the syntax diagram for the definition we will be able to w. 'e the function:



The syntax diagram for the formal parameters is as follows:



Since the body is just some compound statement containing the arithmetic of the function, this gives

```
FUNCTION POWER(X:REAL;Y:INTEGER) : REAL;
  VAR I,POW : INTEGER;
BEGIN
  POW := 1;
  FOR I := 1 TO Y DO POW := POW*X;
  POWER := POW;
END
```

So to find the value of 3 we have to issue the command

```
POWER (3,4)
```

However, this will only work if the value of Y is larger than 0. A more complex function is required if we are to allow negative values and this is shown in the following Pascal program to display a table of values from 1 to 10, showing the number, its square, its cube, its reciprocal and its inverse square:

```
 10 PROGRAM TABLE;
 20 VAR CT : INTEGER;
 30 FUNCTION POWER(X:REAL;Y:INTEGER):REAL;
 40   VAR I:INTEGER;
 50     POW:REAL;
 60   BEGIN
 70     POW := 1;
 80     IF Y <> 0 THEN
 90       FOR I:=1 TO ABS(Y) DO POW:=POW*X;
100       IF Y <0 THEN POW:=1/POW;
110     POWER:=POW
120   END(* POWER *);
130 BEGIN
140   WRITELN('NUMBER SQUARE   CUBE RECIP.');
150   WRITELN('====== ======   ==== ======');
160   FOR CT := 1 TO 10 DO
170     WRITELN(CT:6, POWER(CT,2):7:1,
POWER(CT,3):7:1,' ', POWER(CT,-1):6:4)
180 END.
```

Notice that inside the function there is a declaration for I. Whenever this is necessary, we call I a local variable. Should there be another I used in the program, perhaps in the place of the word CT, then the local variable would not influence its value in any way. This differs from BASIC, in which using the same variable for two different purposes would be fatal and cause many problems.

The variable CT is called a global variable, since it is effective throughout the program and is defined in the usual declaration section of the program.

It is important when using a function that the parameters should be in the same order as they are in the function declaration, i.e.

POWER (4,3)

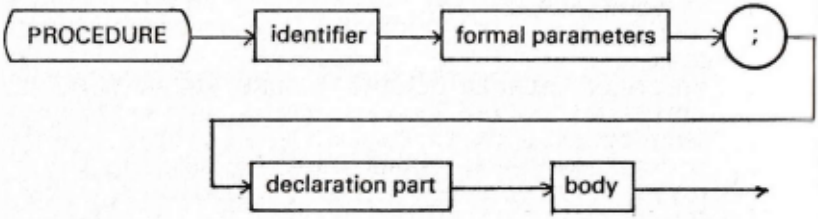
will compute the value of 4^3 and not 3^4. Once the function has been used, the computer forgets the local variables, so they are re-declared each time the function is called.

Procedures

A procedure is similar to a function, but it does not compute and return a value. A procedure is used to break down a program into a series of separate parts, enabling the complete program to

become a sequence of calls to these subroutines. A typical example of where a procedure might be used is when a main menu appears on the screen and the user is offered a choice of several options. Here, each option could be contained in its own procedure, and even the menu itself could be written in such a way.

The syntax of a procedure declaration is very similar to that of a function and can be seen in the following diagram:



Procedures are placed before the main program together with any functions that are being used. To demonstrate their use, here is a version of the program we wrote earlier which gave a choice of either the surface area or volume of a cube, this time using procedures.

```
 10 PROGRAM CUBE;
 20 VAR
 30   DECISION:INTEGER;
 40   LENGTH:REAL;
 50 PROCEDURE CHOICE (VAR DECISION:INTEGER; VAR
LENGTH: REAL);
 60   BEGIN
 70     WRITELN ('FOR A CUBE, DO YOU WISH TO FIND
');
 80     WRITELN('1...SURFACE AREA');
 90     WRITELN('2...VOLUME');
100     WRITELN('3...END PROGRAM');
110     WRITE('CHOICE=');
120     READ(DECISION);
130     WRITE(LENGTH OF SIDE =');
140     READ(LENGTH)
150   END(*CHOICE*);
160 PROCEDURE SAREA(L:REAL);
170   VAR
180     AREA:REAL;
190   BEGIN
200     AREA:=L*L*6;
210     WRITELN('AREA = ',AREA:8:2)
```

```
220    END (*SAREA*);
230  PROCEDURE VOLUME(L:REAL);
240    VAR
250      VOL:REAL;
260    BEGIN
270      VOL:= L*L*L;
280      WRITELN ('VOLUME = ',VOL:8:2)
290    END(*VOLUME*);
300  BEGIN (*MAIN PROGRAM*)
310    CHOICE (DECISION,LENGTH);
320    WHILE DECISION <> 3 DO
330      BEGIN
340        IF DECISION = 1 THEN SAREA(LENGTH) ELSE
VOLUME(LENGTH);
350        CHOICE(DECISION,LENGTH)
360      END
370  END.
```

This program highlights several features of the use of procedures. By using this modular design it is easier to amend any particular subsection and to add or delete routines. For example, we could decide to add to the menu the perimeter of the edges, and this would simply involve a small amendment to the DECISION routine and the composition of a new routine to perform this task. Notice how short the main program becomes and consequently how easy it is to understand.

VOLUME and SAREA are ordinary procedures which accept the value of the parameter and transfer it to a variable called L. Parameters like this are called value-parameters since they take the value of the parameter when the procedure does not return a value and so we have to be a little more clever to alter the value of global variables. To perform this task we have to use a new type of parameter known as a variable-parameter because it can receive a new value during the execution of the procedure. It is represented in the procedure declaration by the word VAR being placed in front of it, as can be seen in the definition of the procedure CHOICE.

This is quite a subtle technique and demands more explanation than these pages can afford. (See the Bibliography for books that deal with it in greater detail.) Be prepared to make several errors before you finally master this technique, but do not despair, for it is extremely powerful when you have grasped its proper use.

CHAR

All the variables we have used up to now have been numerical, either integer or real, but very often when writing programs in Pascal we will want to use characters. Pascal provides the data type CHAR for this purpose. A variable declared to be of this type can take the value of any character in the computer's character set.

We can define character constants as follows:

```
ASTERIX = '*'
BLANK = ' '
```

And we can define variables of type CHAR with statements such as

```
VAR A,B,C,D: CHAR
```

When assigning values to character variables it is important that you place the data within single quotes. For example,

```
A:= 'I'
B := BLANK
C := 'R'
```

Characters can be used in exactly the same way as numbers in Boolean lists with the comparison operators being used to compare them. This however limits us to the use of single characters, and since the normal use of characters is to form words we need to be able to handle such objects. Unfortunately, Pascal makes life a little more difficult than BASIC since there is no equivalent to the string. It is therefore necessary to use an array.

Strings and packed arrays

When using packed arrays (which are the same as ordinary arrays except that the data is compact) to represent a string, we have to state the number of characters that the array will contain. For an ordinary array a declaration such as

```
LIST:ARRAY[1..10] OF CHAR;
```

will form a structure in which LIST[1] contains a character, LIST[2] another, and so on. With a packed array it is possible to regard these 10 letters as a word. Thus a declaration such as

```
COMPOSER : PACKED ARRAY [1..10] OF CHAR;
```

allows the value of COMPOSER to be assigned with statements such as

```
COMPOSER:='BACH      '
COMPOSER:='BEETHOVEN '
```

Notice that the value must always contain 10 characters so the word must be padded out with extra spaces.

One advantage of this is that it allows the value to be tested in one step rather than having to compare each individual character. In other words, it is possible to see if BACH>BEETHOVEN without having to compare the first element and then the second, since the first elements are the same.

Here is a short routine to read in the word, WORD, from the keyboard:

```
CONST
  WORDLENGTH = 10;
VAR
  WORD:PACKEDARRAY [1..WORDLENGTH] OF CHAR;
  I:INTEGER;
BEGIN
  WORD := '          ';
  I := 1 ;
  REPEAT
    READ(WORD[I]);
    I := I+1 ;
  UNTIL (I=11) OR (WORD[I-1] = ' ')
END
```

Other useful commands

One of Pascal's most powerful commands is the CASE function. This can be used to perform a certain statement depending on the value of a variable, instead of having to use multiple IF ... THEN commands. The following two programs segments are identical:

```
IF DAY = 1 THEN WRITE ('SUNDAY')
 ELSE IF DAY = 2 THEN WRITE ('MONDAY')
  ELSE IF DAY = 3 THEN WRITE ('TUESDAY')
   ELSE IF DAY = 4 THEN WRITE ('WEDNESDAY')
    ELSE IF DAY = 5 THEN WRITE ('THURSDAY')
     ELSE IF DAY = 6 THEN WRITE ('FRIDAY')
       ELSE WRITE ('SATURDAY')


CASE DAY OF
   1 : WRITE('SUNDAY');
   2 : WRITE('MONDAY');
   3 : WRITE('TUESDAY');
   4 : WRITE('WEDNESDAY');
   5 : WRITE('THURSDAY');
   6 : WRITE('FRIDAY');
   7 : WRITE('SATURDAY')
END
```

In the CASE statement, depending on the value of DAY, the program will follow the appropriate course. This is much neater than the first example, and is always preferable.

Data types

Finally, a quick mention for Pascal data types. So far we have encountered INTEGER, REAL, BOOLEAN, CHAR, ARRAY, and PACKED ARRAY, but this is not the end of the story. Pascal is extremely flexible since it allows the user to define his own types of data structure, opening up a whole new area of programming.

In the above example the variable DAY could have been declared with the statement

```
DAY : 1..7
```

instead of

```
DAY : INTEGER
```

and then the computer would only have allowed DAY to hold the value 1,2,3,4,5,6 or 7.

Conclusion

Pascal is a very good second language to learn. It has a great advantage over BASIC in that it can produce well designed, carefully planned, structured programs, and if you can program in this way then your algorithms will operate quickly and efficiently. Learning Pascal is bound to improve your programming technique.

6

Pilot

Introduction

Unlike all the other languages discussed so far, PILOT is not a general-purpose language. The name is an acronym of 'Programmed Inquiry, Learning Or Teaching', and the language was developed at the University of California in 1968 as a specialist tool for the development of Computer Assisted Learning (CAL) software.

Unlike Pascal and FORTH, PILOT is very good at handling and manipulating strings, and it is therefore a good language for any application involving large quantities of text. In this chapter we will first consider the basics of the language, then show how what we have learned can be used to generate a simple CAL program, and finally take a look at another use of PILOT – the construction of text adventures.

1. PILOT programming techniques

TYPE statement

The 'TYPE' or 'T' statement is used in PILOT to produce textual messages on the screen. For example,

```
T: THIS IS A MESSAGE DEMONSTRATING
T: THE USE OF THE TYPE STATEMENT.
```

We can see that unlike all other output commands found in alternative languages, the PILOT TYPE statement does not require a delimiter. Therefore any character which can be entered at the keyboard can be displayed using this command. For example,

```
T: HELLO "!1:;,
```

all the characters between the first colon and the return, which signals the end, will be displayed. It is also possible to produce a blank line by using a T with nothing after the colon.

Remarks

As with any language, it is often important to include useful comments within a program, that are not to be executed by the computer. This can be achieved by using the 'REMARK' or 'R' statement.

Throughout the first part of this chapter we will construct a CAL-type program which could be used to test a student's knowledge of the capitals of the world. Using the REMARK and TYPE instructions, the beginning of the program would be as follows:

Example 1

```
R: CAPITALS QUIZ
T:         CAPITALS OF THE WORLD
T:         ====================
T:
T: FOR EACH OF THE FOLLOWING COUNTRIES
T: YOU MUST IDENTIFY THE CAPITAL CITY.
T:
```

ACCEPT

We have seen that output can easily be achieved by using the TYPE statement. However, any program designed for an independent user will require some form of input. In PILOT this is achieved by using the 'ACCEPT' or 'A' statement. For example,

```
A:
```

When PILOT encounters an ACCEPT statement, the normal execution will stop and the computer will wait for a word or sentence to be entered at the keyboard. This input will be automatically displayed on the screen, and the normal sequence will be resumed when the ENTER key is pressed.

MATCH statement

The most important statement in the PILOT language is the 'MATCH' or 'M' statement. This is used to compare a statement input by the user, via an ACCEPT statement, with various words contained in the body of the MATCH.

For example, consider the problem of checking the answer submitted by a student, in answer to the question:

```
T: WHAT IS THE CAPITAL CITY OF RUSSIA?
```

The answer could be checked by using the MATCH

```
M: MOSCOW
```

If the sentence input by the student contained the word 'MOSCOW' an internal flag would be set to 'YES'; if the sentence did not contain the word 'MOSCOW' the flag would be set to 'NO'.

At first glance it is very easy to underestimate the power of the MATCH statement, comparing it to the BASIC line:

```
IF A$="MOSCOW" THEN LET B$ = "YES"
```

In fact there is no comparison, as the BASIC statement will only work if the input variable A$ is exactly "MOSCOW". The MATCH statement, however, will scan the input sentence, looking for the string 'MOSCOW', and therefore each of the following would be recognised:

```
MOSCOW
THE CAPITAL IS MOSCOW
MOSCOW I THINK
```

When the MATCH has been performed and the flag has been set, any future statements can test the flag by using a 'Y' or 'N' together with the instruction code. For example,

```
TY: MOSCOW IS CORRECT
TN: WRONG. THE CORRECT ANSWER IS MOSCOW!
```

This checking of the match flag can continue until another match is performed, thus changing the status of the flag.

Branching

It is very unlikely that we will write a program which will always be executed in exactly the same way. And it is likely that we will require some facility which will enable us to jump from one part of the program to another.

This is achieved by using the 'JUMP' or 'J' statement, together with a condition 'Y' or 'N', and a label indicating the precise position in the program from which execution is to be continued.

The rules governing labels will depend on the version of PILOT being used. The Spectrum version allows the use of six characters (letters or numbers only), with the first character being a letter. Of the following, the first two are legal labels, the second two illegal:

```
* PART1      (legal)
* SECTOR     (legal)
* 1PART      (illegal: number first)
* END.2      (illegal: full stop included)
```

From these examples we can see that a label is prefixed by an asterisk and, unlike all other PILOT statements, does not take a colon. This is because a label is purely a marker pointing to a position within a program; therefore it does not have to be interpreted and executed in the same way as a normal line of program.

Example 2

We can employ labels and jumps in a program in order to keep asking a question until the user enters the correct answer.

```
*LABEL1
 T: CAPITAL OF SPAIN?
 A:
 M: MADRID
TY: MADRID IS CORRECT
JN: LABEL1
```

Although this program is syntactically correct, it is not very useful in practice, since if the user had no idea of the correct answer, a stalemate would ensue. This could be overcome by using the 'label-less' jump statement and an answer-counting routine.

Answering a question over and over again is common in any CAL program, and accordingly PILOT provides the programmer with a special statement:

```
J: 2A
```

This causes the program to jump back to the last ACCEPT statement so that the user can submit an alternative answer.

Subroutines

Although PILOT doest not use subroutines as such, it does have a 'USE' or 'U' statement, which will cause the program to branch to a section of code identified by a label and ending in the 'END' or 'E' statement. Unlike the JUMP statement, USE remembers the position that it came from and is able to return to this line on completion of the block. For example, a typical routine to congratulate the user could consist of the following:

```
*RIGHT
T: **********
T: *WELL DONE*
T: **********
T:
T: PRESS 'ENTER TO CONTINUE'
A:
E:
```

This routine could be called from anywhere within the program by using a line such as:

```
UY: RIGHT
```

Further matching (AND and OR)

We have already mentioned the importance and power of the MATCH statement. In its basic form, however, it is only capable of searching a sentence for a particular string, which is generally

unsatisfactory as we may want to match an input with several different strings. This can be achieved in one of two ways, using either the 'AND' or the 'OR' statement.

If we want to scan an input sentence for a string consisting of two or more substrings, this can be done by using the AND (*) statement. For example,

```
M: CHARLES * DICKENS
```

This match could be used to scan an input sentence for the two names 'CHARLES' and 'DICKENS', and the match flag would only be set to 'YES' if both names existed.

Similarly, we may want to search an input sentence for one of several strings, and this can be done by using the OR (,) statement. For example,

```
M: CAT,DOG,MOUSE
```

The match flag would be set to 'YES' if one of the three substrings existed.

When using AND you may encounter one of the few quirks of using the MATCH statement: the substrings must be in the same order in the MATCH statement as they occur in the input sentence. For example,

```
MATCH = CHARLES * DICKENS.
INPUT – CHARLES FREDERICK DICKENS ——› TRUE
INPUT – DICKENS CHARLES ——› FALSE
```

The simplest solution to this problem is to use an OR to include both possible orderings. For example,

```
M: CHARLES * DICKENS , DICKENS * CHARLES
```

This is satisfactory when there are only two substrings. If there were more than two this method would be somewhat cumbersome, and would best be replaced by using a number of MATCH statements, as follows:

```
MY: A
MY: B
MY: C
MY: D
TY: MATCH FOUND
```

Once a MATCH fails, the rest of the statements will not be executed.

Blanks (%)

Consider the problem of matching an input sentence to find the word 'GET'. If the sentence contained the word 'FORGET', then the match would be successful. This should not be the case, however, as we are looking for the word 'GET' as opposed to the string of characters made from the letters G, E and T. This problem can be overcome by using the fact that a word will always be surrounded by spaces, which can be represented in the MATCH statement by using a '%' sign. For example,

```
M: %GET%.
```

Wildcards (?)

Any good CAL program will allow spelling mistakes or inconsistencies to be made in an input, yet still provide a satisfactory result in the match statement. This effect can be produced by using the wildcard character, '?', which means anything. If the match is

```
M: DIS?
```

DISK or DISC would be recognised as correct.

Modifiers

Some versions of PILOT allow an 'S' or 'spelling' modifier to be appended to the normal MATCH instruction. For example, if the match is

```
MS: HANDKERCHIEF
```

this spelling modifier will allow certain incorrect spelling to be matched. The range of error detected by such a system will depend on the implementation, but most will detect single character and pair transposition errors:

HANKERCHIEF —› TRUE
HANKERTCHEIF —› TRUE
HANKRCHIEF —› FALSE

Number handling

As mentioned previously, PILOT is not a 'number-crunching' language, as it was designed specifically for the manipulation of textual data. However, having said this, there will come a time when it is necessary to perform some simple mathematical process using PILOT. In PILOT all arithmetic is performed in a 'COMPUTE' or 'C' statement, using the following operators:

Operator	Action
+	addition
—	subtraction
*	multiplication
/	division
()	brackets

Variables

The use of variables within PILOT is similar to that in BASIC, in that they can be used at any time without being pre-defined. The only limitation is that variable names can consist of one letter only, and this restricts us to a total of 26 possible variables.

The COMPUTE statement follows the same syntax rules as the others and can have an associated modifier such as 'Y', 'N', etc. The following are all legal COMPUTE statements:

```
C: S=S+1
C: T=T*4
C: S=(T+S)/4
```

The use of number variables within a PILOT program opens up numerous possibilities, as the variable name could be used with input and output statements, and as part of a modifier of any PILOT statement.

Consider the problem of using a variable 'S' to maintain the score obtained in a quiz. When it becomes necessary to print the score we might attempt to use:

```
T: YOUR SCORE = S
```

This, however, is unsatisfactory as 'YOUR SCORE = S' will simply be reproduced on the screen. We require a way of informing the computer that 'S' is to be considered as a variable. This is achieved by using the '#' characters as shown below:

```
T: YOUR SCORE =#S
```

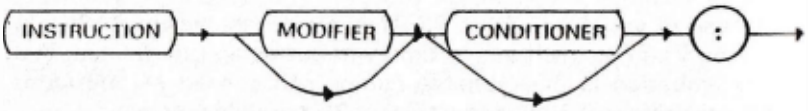
The '#' character can also be used within an ACCEPT statement to enable the user to input a number which will be assigned to some variable. For example,

```
A: #S
```

It should be mentioned at this point that arithmetic cannot be performed in a TYPE or ACCEPT statement; all calculations must be evaluated in COMPUTE lines.

Relational conditioners

All PILOT statements obey the following syntax diagram:



As can be seen from the diagram, the modifier and the conditioner are both optional. We have already seen that the conditioner can be 'Y' or 'N', reflecting the condition of the match flag, or an integer in the range 1 to 9, to test the answer-counter.

It is also possible to use variables and the following relational operators in the construction of conditioners:

Operator	Action
=	equals
<	less than
>	greater than
<=	less than or equals
>=	greater than or equals
<>	not equal

For example,

```
T(S<10) : YOU HAVE SCORED LESS THAN 10
T(O>4) : START.
```

In each of the above examples, the statement will only be executed if the condition in brackets is 'TRUE'.

String variables

PILOT is similar to BASIC in many ways; in fact the rules pertaining to variables and arithmetic are very similar, as is the general structure of programs in the two languages. One area in which the languages differ, however, is in the use of string variables. Any string variable used in PILOT is indicated by using a '$' sign, however unlike BASIC the '$' sign comes before the variable name. A string can be assigned by using the COMPUTE statement:

```
C: $NAME = HELLO
```

or by using the ACCEPT statement:

```
A: $$NAME
```

The first dollar sign in the ACCEPT statement is simply to indicate that what is being entered should be considered as a string variable. It plays a similar role to that of the '#' character used with a numerical variable. The same symbol is employed in TYPE statements to display the value of a string variable:

We have now considered the basics of the language and are in a position to implement these ideas in the solution of two problems, both of which are highly textual.

2. CAL programming

In the following program we will consider how the techniques considered in this chapter can be used to produce a computer assisted learning (CAL) program dealing with capital cities. The subject matter is not important, as it is the underlying principles with which we are concerned, and these will be the same in all cases.

```
      C: S=0
      C: C=0
     TH:
      T:    CAPITALS OF THE WORLD
      T:    ====================
      T:
      T:    FOR EACH OF THE FOLLOWING
      T:    COUNTRIES YOU MUST IDENTIFY THE
      T:    CAPITAL CITY.
     PA:    200
     CH:
      T:    WHAT IS THE CAPITAL OF RUSSIA?
      T:
      A:
      C: C=C+1
      M: MOSCOW
     TY: WELL DONE, MOSCOW IS CORRECT.
     CY: S=S+1
     JY: Q2
J(C=2): Q2
      T: HARD LUCK, ONE MORE TRY.
      T:
      J: @A
    *Q2
      T:
      T: 'ENTER' TO CONTINUE
      A:
     CH:
      T: WHAT IS THE CAPITAL OF SPAIN ?
      T:
      A:
      C: C=C+1
      M: MADRID
     TY: WELL DONE, MADRID IS CORRECT
     CY: S=S+1
     JY: Q3
T(C=2): CORRECT ANSWER = MADRID
J(C=2): Q3
      T: HARD LUCK, ONE MORE TRY
      T:
      J: @A
    *Q3
      T:
      T: 'ENTER' TO CONTINUE
      A:
     CH:
      C: C=0
      T: WHAT IS THE CAPITAL OF ITALY ?
```

```
      T:
      A:
      C: C=C+1
      M: ROME
     TY: WELL DONE, ROME IS CORRECT
     CY: S=S+1
     YJ: END
T(C=2): CORRECT ANSWER = ROME
J(C=2): END
  *END
    CH:
     T: QUIZ OVER. SCORE = #S
```

The program is short, but it demonstrates most of the facilities covered in this chapter. Although it is very repetitive, improved techniques could be developed with some thought and a more detailed understanding of the language.

3. Text adventures

Another area in which PILOT is very useful is in the formation of text adventures. This style of game has a huge following and there are large rewards to be had by anyone who can master the techniques involved.

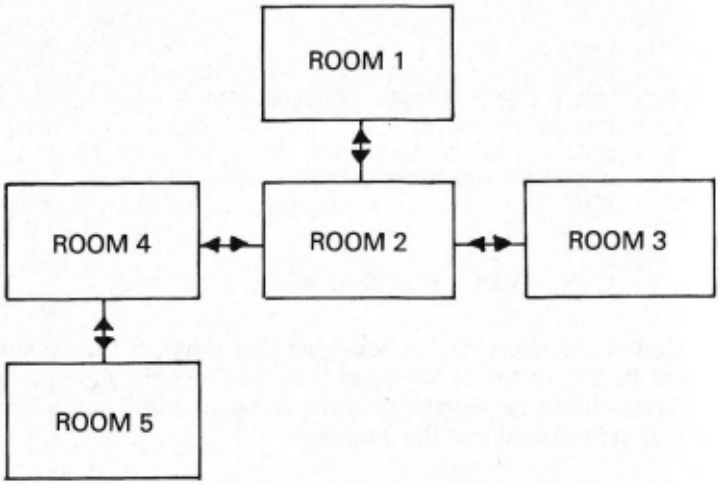
Below we show you how to design a text adventure and then implement it using PILOT.

Stage 1

The first stage of designing an adventure is to decide on the scenario, i.e. where the adventure takes place and what the final objective is. In our example the adventure takes place in a small bungalow and the objective is to find the combination of a safe-deposit box containing your inheritance from your late uncle D.K. Worth.

Stage 2

Having decided where the adventure takes place, we are now in a position to construct a map showing the individual locations and the connecting routes.



Stage 3

Now that the objective is clear and a suitable map has been created, we can consider the details of each individual room. Each room will require a description to be displayed on the screen, together with a list of objectives which must be accomplished. This information is best displayed in a table as shown below.

Room	Description	Objectives
1	This is the master bedroom. To the south, you can see a door leading to a hallway. In the corner of the room, there is a computer. Attached to the computer is a disc drive and a monitor.	The computer must be turned on and a disc inserted into the drive; the combination will then appear on the screen.
2	You are standing in a hallway, to the north and east there are doors, and to the west you can see a stairway.	None.

3	You are in a bathroom containing a bath, sink, toilet and cabinet. To the west there is a door leading to the hall.	Inside the cabinet there is a key which must be acquired.
4	You are in the lounge. To the east are the stairs and to the south is the kitchen. In the corner of the room is a piano.	None.
5	This room appears to be the kitchen. There is a cupboard, a cooker and a sink. The only exit is to the north.	The cupboard is locked and contains a disc.

Stage 4

We have now completed the preparation, and are in a position to write the code for the adventure:

```
    CH:
     T:     TEST ADVENTURE
     T:     ==============
     T:
     C: D=0
     C: K=0
     C: O=0
    PA: 200
*ROOM1
    CH:
     T: THIS IS THE MASTER BEDROOM
     T: TO THE SOUTH, YOU CAN SEE A
     T: DOOR LEADING TO A HALLWAY.
     T: IN THE CORNER OF THE ROOM
     T: THERE IS A COMPUTER, ATTACHED
     T: TO THE COMPUTER IS A DISC
     T: DRIVE AND MONITOR
     T:
     A:
     M: SOUTH
```

```
     JY: ROOM 2
      M: SWITCH
     MY: COMPUTER
     TY: THE COMPUTER COMES ON, AND
     TY: YOU ARE REQUESTED TO PLACE A
     TY: DISC IN THE DRIVE.
      M: PUT
     MY: DISK
TY(D=1): THE COMBINATION IS 427106
JY(D=1): END
      T:
      T: YOU CANNOT DO THAT
      J: ROOM 1
 *ROOM2
     CH:
      T: YOU ARE STANDING IN A HALLWAY,
      T: TO THE NORTH AND EAST THERE
      T: ARE DOORS, AND TO THE WEST YOU
      T: CAN SEE A STAIRWAY
      T:
      A:
      M: NORTH,EAST,WEST
     JM: ROOM1,ROOM3,ROOM4
      T:
      T: YOU CANNOT DO THAT
     PA: 1
     PA: 200
      J: ROOM2
 *ROOM3
     CH:
      T: YOU ARE IN A BATHROOM, CONTAINING
      T: A BATH, SINK, TOILET AND CABINET
      T: TO THE WEST THERE IS A DOOR
      T: LEADING TO THE HALL
      T:
      A:
      M: WEST
     JY: ROOM2
      M: OPEN
     MY: CABINET
      T:
     TY: THE CABINET OPENS AND YOU SEE A KEY
     AY:
     TY:
     MY: GET
     TY: YOU HAVE THE KEY
     CY: K=1
     TN: YOU CANNOT DO THAT
```

```
     PA: 1
     PA: 200
      J: ROOM3
*ROOM4
     CH:
      T: YOU ARE IN THE LOUNGE. TO THE
      T: EAST ARE THE STAIRS AND TO THE
      T: SOUTH IS THE KITCHEN. IN THE
      T: CORNER OF THE ROOM IS A PIANO.
      T:
      A:
      M: PIANO
     TY: DO YOU CALL THAT
     TY: PLAYING THE PIANO!!!
    PAY: 1
    PAY: 200
     JY: ROOM 4
      M: EAST, SOUTH
     JM: ROOM2,ROOM5
      T:
      T: YOU CANNOT DO THAT
     PA: 1
     PA: 200
      J: ROOM 4
*ROOM 5
     CH:
      T: THIS ROOM APPEARS TO BE THE
      T: KITCHEN. THERE IS A CUPBOARD,
      T: A COOKER AND A SINK. THE ONLY
      T: EXIT IS TO THE NORTH.
      T:
      A:
      M: NORTH
     JY: ROOM 4
      M: OPEN
     MY: CUPBOARD
      T:
TY(K=1): THE CUPBOARD IS OPEN
TY(K=1): YOU SEE A DISC ON THE SHELF.
CY(K=1): O=1
AY(K=1):
MY(K=1): GET
MY(K=1): DISK
CY(O=1): D=1
TY(O=1): YOU NOW HAVE THE DISK
     PA: 1
     PA: 200
     JY: ROOM 4
```

```
    T: YOU CANNOT DO THAT
   PA: 1
   PA: 200
    J: ROOM3
*END
```

In this program we have introduced a number of new facilities which are available in this version of PILOT:

1. The 'CH' command is used to clear the screen, returning the cursor to the top left hand corner.
2. 'PA' is used to instigate a pause. These are used in pairs, in order to remove any 'ENTER' characters contained within the input buffer.
3. The program makes use of the multiple matches, together with the multiple jump statement 'JM'. For example,

   M: EAST,SOUITH,WEST
   JM: ROOM1,ROOM5,ROOM6

   If the ACCEPT statement is matched with the first listed item then control is transferred to the first label, etc.

Conclusion

As with many of the alternative languages for the Spectrum, this version of PILOT is not a full implementation of the original, and various functions such as arrays and answer counters have been omitted. The implementation does, however, contain numerous extra facilities. As a language for text and string handling it is highly recommended.

7

Pseudo-Languages with Specific Applications

Here we consider four of the pseudo-languages available for the Spectrum. These are not full computer languages such as those encountered so far, but are designed with a specific application in mind – normally the production of machine-code routines for games programs.

Games Designer by Software Studios

Games Designer is a program which can be used to produce smoothly animated machine-code games without needing to know a programming language. It is totally menu-driven, which means that all the necessary information about a game is entered via prompts. However, this does mean that the range of games that can be produced is fairly limited. All the possible games are variants on the familiar 'shoot-em-up' theme.

There are eight different basic games types, such as 'Space Invaders' and 'Asteroids'. When the Games Designer program is first loaded there are eight games already programmed into the machine: one of each type; although they are good fun to play initially, the novelty soon wears off and you will want to have a crack at designing your own.

There are 32 sprites available for you to use (a sprite is a collection of pixels which move across the screen in unison and provide the moving or animated characters for the game). Each sprite's shape and colour can be changed at will using the sprite editor facility. You can make sprites appear animated by alternating between two sprites with slightly different pixel arrangements. The kinds of sprite you can define are aliens, ships or laser bases, missiles, shields and explosions: certain sprite numbers are reserved for each of these categories. All the sprites are twelve pixels square, which is one and a half times as large as a normal graphic character.

Once the sprites have been defined you can decide how they

are going to move across the screen. There can be eight different movement patterns for each game; each pattern is made up of a numerical list plotting the movement directions. The patterns are shown graphically on the screen as they are defined, and you can link the patterns together. In other words, when an alien has finished moving along the path dictated by pattern number one, it can be told to move along the trail of, say, pattern number five.

The sound effects that are to be used in the game are designed using a system of graphical slider switches which can be moved up or down on the screen. You can define four different sounds in each game; one each for missiles, bombs, exploding ships and exploding aliens. Each of the five slider switches represent one characteristic of the sound wave to be generated, such as frequency and length. You can test the sound you are producing at any time, and it is possible to produce a large range of interesting noises using the system.

One of the failings of the program is that all the games are played on a plain background (or one with stars). It is not possible to define a landscape for your game; you can only change the basic colour of the background.

The way the game progresses is controlled by the attack waves chart. There are six columns in this chart, controlling such things as the speed of the aliens, which aliens are to be used in a particular attack wave, the score obtained for shooting down an alien from the wave and the maximum number of aliens involved in the attack. Any of those criteria can be altered by using a cursor which moves across the table. In order to produce your attack wave table correctly, it is necessary to refer to the tables in the back of the manual quite regularly, and this can become tedious.

Because of the way the program is designed it is possible to have up to eight different games in the memory at any one time, so you can work on several different projects at once if you so wish. The manual provided with the program is fairly short, and you will probably find that it takes quite a lot of experimentation to be able to design games efficiently. Parts of the manual, such as the section on attack waves, are rather vague and oversimplified, telling you that you can do something but not explaining how.

When you actually come to play a pre-programmed game or one of the games you have designed, you will find they are extremely smooth and can be quite fast. The colour clash between sprites is dealt with fairly efficiently when it occurs, and the general impression is one of a well-presented game. There is a high score table, and the whole character set has been redefined

to a space-age looking set of letters. The game editor is a pleasure to use because it is so well presented. A great deal of attention has been paid to centering items on the screen and presenting the various menus in an appealing format.

You can save the games that you have designed on to tape (or microdrive, if you have the microdrive version) but they cannot be played independently. The main Games Designer program must be loaded in before you can use them, so you cannot use the program to design games for marketing or publication.

Overall, the scope of Games Designer is slightly more limited than some of the other packages available. It is not possible to produce original games or Manic Miner lookalikes, and the 'shoot-em-up' type games produced do not tend to hold the interest for all that long. It is certainly a good program for someone who has only recently been introduced to computing because you can produce your individual results without any knowledge whatsoever of computers; however, you do have to be fairly clever to understand the concise manual!

The Quill by Gilsoft

The Quill is a program package which enables a novice to produce high quality machine-code adventures, which will run independently of the main generator program when they are finished. One of the most important advantages of the Quill is that you are allowed to market any of the adventures you produce independently, and Gilsoft are also prepared to sell your adventures themselves if they are good enough.

The program is basically a large database with an editor and an interpreter system which controls the adventure. Contained within the database are the descriptions of locations, movement tables, a vocabulary list, object list, flags and the so-called 'status' and 'event' tables. The editor allows you to change any of the items in the database or to add new information, while the interpreter uses the information amalgamated in the database to execute the adventure game and to control the detailed progression of the adventure.

In order to produce an adventure, the user first has to design a map of all the locations, decide the initial positions of the objects, and work out how the rooms are interconnected. As the program is completely menu and sub-menu driven, it is useful to have all the information for the database written down before you start so that you do not need continually to switch between menus. The

descriptions of the locations are then entered into the computer and the possible movement directions from each room are typed in. The object names are inserted into the database along with their starting location numbers. Note that it is possible to test the results of your efforts at any time by playing the adventure as far as you have got, but this does involve switching around between menus. However, you can check if your on-screen presentation is correct before you get too deeply involved with the location texts.

All the vocabulary that the computer will ever need to understand is then typed in, including object names. All inputs made during the adventure are checked against the vocabulary database and if a word is not in the computer's dictionary it will be ignored: this is an improvement on most adventures because it is usually the accepted practice to print up 'I do not understand that' even if only one word is not understood. All the vocabulary words are abbreviated to four letters. Adventure messages are stored in another part of the database; these are sentences like 'You are feeling hungry' or 'You are dead', and the computer can be told to print them up in certain circumstances.

The most important parts of the database are the status and event tables which are where the actions relating to a player's input are stored. A typical entry in the event table would take the form:

```
INPUT          CONDITION(S)          ACTIONS ... OK,
```

where the input is the word or words which the player had typed in, and the actions are only carried out if all the conditions are satisfied. The conditions usually involve a flag, room numbers or object number, i.e. PRESENT 1 means only do the required action if object 1 is present in this room. There are 32 flags that can be used in your adventure: some of them are used for specific purposes such as a count of numbers of turns taken so far, while others can be defined by the user: for example a certain flag may be switched off (0) if a door is closed and on (1) if the same door is open. These flags are fairly simple to control and use. An actual entry in the event table might look like this:

```
INPUT       CONDITION                    ACTIONS
EAT APPLE   PRESENT 1(Apple is object 1) DESTROY1
                                         MESSAGE1
                                         OK
```

DESTROY1 makes object 1 (the apple) non existent.
MESSAGE1 prints up message number 1, e.g. 'You are no longer hungry'

It is found that in order to be able to produce every situation that might occur or be necessary in an adventure only a limited range of possible conditions and actions are required. As you get to know the system, you will find that every command can be expressed in terms of a few simple actions.

The status table is very much like the event table: it has the same format, but it is scanned at the beginning of every turn to check for a situation which may have developed during the previous turn. Thus the status table would have entries to check if the player is dead, or hungry, or has taken too many goes. It is these two tables which form the backbone of the adventure system.

When the program is first loaded, there are quite a large number of items already in the database such as standard messages like 'You cannot go in that direction' and 'I do not understand that', and movement commands with a few standardised vocabulary words. Having this pre-programmed data does cut down on the rather boring job of entering vocabulary.

The Quill is very well documented. The manual is divided into three main sections: there is a novice's guide to writing a first-time adventure using the Quill, a detailed and quite technical description of the editor, database and interpreter, and a summary of all the possible conditions, actions, and flags used in the program, for quick reference.

When writing an adventure using the Quill it is necessary to have quite a large amount of written information to enable you to work quickly and efficiently. It is not a good idea to have to keep switching menus to find out what room number a particular room has because this can be very time-consuming. It is far better to have a list of all the room numbers, object numbers, message numbers, vocabulary words and flags written down: each sub-menu has a print option which allows you to get a hard copy of the data from that section on a ZX printer to use for reference. Incidentally, Gilsoft, the company who manufacture the Quill, have recently brought out a companion program called the Illustrator, which allows you to add graphic designs to your locations.

The high standard of adventures produced using the Quill can easily be seen from the number and quality of 'Quilled' adventure games currently on the market. According to the magazine

reviewers, the 'Quilled' adventures are just as good as ones which have been programmed from scratch: all you need now is a good original idea and some challenging problems in an adventure, and all the hard work (the programming) will be done by the computer. The Quill allows people who have no programming experience to write high quality adventures, and it is most certainly an excellent buy!

HURG by Melbourne House

HURG (High-level User-friendly Real-time Games designer) is ideally suited to a person with little knowledge of BASIC who wishes to produce original games of commercial quality. The whole program is based on a network of menus which can easily be manipulated to provide the required configurations using some simple cursor controls. It has the advantage of superb graphics without having to resort to machine code. It adopts a very flexible approach compared with other games-orientated languages in that the format of the game is not restricted. HURG bridges the gap between a good programmer with no imagination and someone with little programming knowledge but a vivid imagination. A complete game can be produced using this package by either editing one of the example games or starting from scratch.

When you first try fiddling around with HURG you will find that it is very easy to get lost in a maze of menus – the easiest way to find your way back to the main menu is to keep pressing <ENTER>. Try loading one of the example games by selecting LOAD GAME, using Q,Z,P and I keys, on the main menu and pressing 0 or N. Now insert the HURG cassette on the other side and press 'play'. Remember the way you selected this option, because that is the way that any option is selected on the various menus. Using the maps provided in the manual, try to find your way around the different menus.

This is a perfectly valid way to become adept at handling the menu, but the best way to familiarise yourself with HURG is probably to write your own simple game. To do this you will have to clear any previous work by selecting RESET HURG from the main menu. You are now ready to design your own game by selecting EDIT GAME. The first thing to try is creating the player character. In order to do this you must select GAME VARIATIONS, NORMAL GAME, PLAYER MENU, in turn. Now select SIZE/ANIMATION and proceed through the options stage by

stage, designing your character as you go. When you reach the collision page choose the colour of your character and then what happens on meeting various ink and paper combinations. For example, if you want your character to crash on meeting some background with blue ink and white paper, select CRASH and position the symbol at the intersection of ink1 and paper7, then press N. To choose another symbol press <SPACE>.

On finishing this stage you will enter a GENERAL menu. Select REGENERATION and follow through the various stages, which are virtually self-explanatory. The SPECIAL EVENT stage allows you to determine what happens if your player collides with a certain ink and paper combination. The only complicated option on this page is the variation option. If this is your first game then it might be wise to leave this until a later date, when you have got a simple game in operation – so there is no need to read the next paragraph unless you are interested.

A variation is when something drastic happens to the game. For example, when a pacman eats a power-pill several aspects of the game are changed for a certain length of time. This is catered for by the variation option: you select an undefined variation number and specify the duration. You then come back later, having created the rest of the game, and select the required variation from the GAME VARIATIONS menu. You then alter the parts of the game which need to change.

The next step is to define the other objects that need to move on the screen. This is done by selecting NORMAL GAME, OBJECTS MENU, and then choosing the object number that you are going to design. You can then create an object in the same way as you created the player character, with only one difference: there is an extra stage, MOVEMENT PATTERN. This stage allows you to describe the motion of the object in many ways. It is probably one of the most powerful parts of the program, because alongside each type of movement is a 'weighting'. This means that you can force the object to perform one type of motion more than another. The 'weightings' are ratios between the various motions. For example, UP:002, DOWN:001 means that the object is twice as likely to go up as it is to go down. If you then added MOVE TOWARDS PLAYER:010, the object would be five times as likely to move towards the player as it would be to move up. Another interesting feature is the USER-DEFINED PATH, which enables you to define a set of movements. If you require this option then you must already have created some paths in the DEFINE PATHWAYS section. So if you select a 'weighting' greater than 000, you will automatically enter the PATH LINKER, where

you will be expected to link four, not necessarily different, paths together to produce a long path of 200 moves. Note that if you give other movements a 'weighting' as well as this, the object will not stick exactly to its set path.

Your next task is to decide what the fire button will do in your game. Enter the FIRE BUTTON ACTION menu and select the desired option. If you choose SHOOTS then you will automatically be asked to design the bullet, which is done in exactly the same way as the other objects. If you choose JUMPS then you will be required to design the jump pattern and decide how far the player can fall without dying.

Now to add the finishing touches: define the scoring system in SCORING, determine when a player starts a new screen in NEW FRAME CONDITIONS, design a title screen with instructions, including the game characters, in TITLE PAGE, and load a background from tape in LOAD BACKGROUND. The background must be saved as a SCREEN, having been drawn in BASIC or by a utility such as MELBOURNE DRAW.

One of the most helpful parts of the manual is the diagram of the menus. Most of the menus allow you to save the information covered by the particular menu, so that you do not have to design the whole game in one go.

White Lightning by Oasis Software

This package combines the speed of a graphics FORTH with the simplicity of BASIC. However it is really only for serious users who are prepared to learn a new language in order to produce sophisticated games of commercial quality. A knowledge of FORTH is desirable (see Chapter 2 above), but not necessary. The package includes an impressive demonstration program and a powerful sprite generator, which has a more than adequate library of arcade graphics – from space invaders to donkey kong. Games written using White Lightning can be compiled and then run independently from the master program, so there are no restrictions on software created on White Lightning.

White Lightning is based on the standard fig-FORTH, but its great advantage is the IDEAL sub-language which is incorporated into the operating system. IDEAL stands for 'Interrupt-Driven Extendable Animation Language' and provides the complex sprite-handling capability of White Lightning. For those of you who have never used FORTH, its main advantage is that it produces compiled machine code, which gives it its great speed.

Having said that, it does not follow the same structure as Pascal, but is built up from many simple commands into very complex ones. There are several commands in White Lightning which cater for interaction between itself and BASIC. This means that you can either use White Lightning as a subroutine in your BASIC program or use BASIC as a subroutine in your FORTH-based program. This allows a relative newcomer to use some simple White Lightning routines to liven up his BASIC program. You can then progress from there in slow easy stages until you write your whole program in FORTH.

IDEAL is the real key to White Lightning. It opens up new horizons in the speed and complexity of on-screen animation. It includes tricky routines like rotations and enlargements, and means that smooth flicker-free graphics are a real possibility. White Lightning provides a real alternative to learning machine code on the Spectrum, but we would stress that if you do not know FORTH it would be wise to read up on the language before attempting any complex programming. Another major feature of IDEAL is its handling of interrupts. This allows you to execute a procedure at regular intervals: e.g. the scrolling background in Defender needs to be continually updated.

The first thing to look at is the demonstration program, which gives you some idea of the effects that you will be able to achieve, and we will describe some of these below.

The lightning is produced using the very quick mirror facility. Smooth single pixel background scrolling using interrupts can be seen at the top of the screen. The words 'White' and 'Lightning' are rapidly character scrolled left and right respectively.

The wheels of the train are several sprites switched very quickly. The track is pixel scrolled, and by using a delay loop the train's apparent speed can be altered.

Some of the spiders are vertically scrolled at various rates using interrupts, while others are animated by switching between sprites.

The invaders are scrolled with wrap within individual windows, while the central invader is pixel scrolled using interrupts. You should note that the speed of the invaders in the circle does not affect the speed of the central invader – this is a property of interrupt-driven routines.

The clockwork toys are animated using two different sprites and each sprite is individually placed on the screen. The whole manoeuvre makes extensive use of DO loops.

The moving coloured stripes exhibit the vertical attribute scroll with varying timing loops. The random number generator is

demonstrated in the form of the border colour.

The car race shows clearly the difference between the three types of horizontal scroll: 1 pixel, 4 pixels and 8 pixels at a time, from top to bottom.

The television illustrates the possibility of windows on the screen. Horizontal scroll at different rates with wrap is seen with the dancer, the duck and the rocket.

The three spaceships combine many features, and this screen clearly shows the complexity of the routines that can be handled. Several different types of scroll are occurring simultaneously.

The bouncing man shows the XOR logical operation, which means that the man can move through the text without obliterating it. In this case it was necessary to disable interrupts to remove a slight flicker.

The various orientations of the invaders were created from the original invader using the sprite generator program. Thus animation is achieved by switching sprites in succession. Interrupts are used again to scroll the central invader smoothly across the screen.

The screen full of crabs demonstrates the speed of the graphics routines and the random number generator.

The bouncing ball is a series of sprites which give half character resolution, and the listing given on p. 75 of the manual is well worth studying.

The rotating radar dish on top of the space ship is achieved by using eight different sprites. On take-off you should notice that the explosion does not obliterate the space ship. The space ship is then scrolled vertically, while the ground is scrolled sideways.

The rotating balls again make use of several sprites place on the screen in order to give the effect of rotation. The credits were simply scrolled slowly up the screen.

Before you can write your own game, you must use the Sprite Generator program to design the sprites that you require. It is suggested that you make good use of the extensive library of graphics which is excellent. This program provides a good introduction to the main program by making you use commands like GET and PUT which are very important words in White Lightning. Pp. 83-8 of the manual provide a useful summary of the function keys, which require a little practice before you get used to them.

When you first load the program you must execute a COLD START (press C), but if for any reason you BREAK the program or an error occurs then you must type GOTO3 and execute a WARM START. To start with you should leave the buffer size as 256

(press N) – you only need to change this if you are trying to save a few extra bytes during the latter stages of program development. The character square is the area used to create and edit sprites one character at a time. The sprite screen is the large square on which sprites are developed. A character can be designed in the character square using the cursor keys, the 9 key to set a bit and 0 key to reset it. Press the Q key followed by Y to clear this grid. To clear the sprite screen press Symbol Shift and Q. Move the X and Y cursors to 1,1 by using Symbol Shift and the cursor keys. Press Z to call up a character from the library – try 75. You can see that the character has been placed at 1,1 on the sprite screen. Press K and the character will appear in the character square where you can edit it.

To store your new character in memory, the GET command must be used. You can only GET a sprite from the sprite screen – so first you must move it, using the J key. Your character will then appear on the sprite screen at the position pointed to by the X and Y cursors. This can now be stored by pressing S followed by 1 to indicated the sprite number and then G to GET your sprite into memory. To prove that your sprite is in memory, clear the sprite screen, press S followed by 1 and then P to PUT it on to the screen, choose option 1, and it should appear.

Once you have got the hang of this, try experimenting on your own. Now try out all the other function keys – fiddling around on your own is always the best way to get used to a new program. If you get stuck then consult the manual and if this can not help then press BREAK, GOTO3 and execute a Warm Start.

Having created your masterpieces you will need to SAVE them to tape ready for use with the main program. To do this type Symbol Shift and S. If you wish to edit these at a later date, they can be loaded with Symbol Shift and J.

FORTH is neither an interpreter nor a compiler, but combines the best features of both to give a very fast, high-level language. It has an interactive interpreter and executes almost as fast as machine code. FORTH uses a computation stack and uses Reverse Polish Notation. This is very different from anything in BASIC and takes a while to get used to. FORTH tends to use integer arithmetic and this version is no exception, although 32-bit precision is available if necessary. White Lightning incorporates over a hundred extensions to the standard Fig-FORTH, including the IDEAL sub-language and all the BASIC, high resolution commands. The idea of FORTH is to add new commands to the vocabulary, using previously defined commands. The complex commands needed to carry out what you require can be built up in

this way. Fully structured programming techniques are essential in FORTH, using BEGIN ... UNTIL, DO loops, etc. An editing system similar to that of Spectrum BASIC is used to build up source code in 22 different 'pages', each with 8 lines and 64 characters per line. This does not limit the size of the program, because the source code can be compiled and more can then be added. It is important to note that the first five screens store the machine code required for interrupt routines. So if you wish to make use of interrupts in your program, do not alter these pages.

White Lightning is made up of FORTH words which are always separated by spaces. A FORTH word must be less than 31 characters long and must not contain any spaces. To see how FORTH works let us look at an example. Type the following (all separated by spaces):

```
1 2 + 3 *
```

Then press <ENTER>.

This puts the number 1 on the stack and then puts the number 2 on top of it. The addition symbol causes the top two numbers on the stack to be removed and replaced with their sum. Therefore the stack now holds the number 3. Another number 3 is then placed on top of this 3 and the multiplication symbol removes them and replaces them with their product, 9. To see the result of these calculations, i.e. the top number on the stack, type a full stop (symbol shift and M) and press <ENTER>. '9 OK' should appear on the line below. 'OK' is always printed at the successful completion of a FORTH instruction. Read through the first few pages of section 2 of the manual and try out various FORTH words shown there. Most of them are fairly self-explanatory.

Obviously it is possible to write some quite complex routines in this way, but the computer does not remember what you typed, so you cannot execute it again without retyping the whole line. This is where 'colon defintions' come in. To define a new FORTH word type a colon followed by a space and then the name of the new word you want to define. You can now type in a line or series of lines of FORTH words. Just press <ENTER> to move on to a new line. To terminate the definition type a semi-colon followed by <ENTER>. Try typing the following definition:

```
: FRED DUP * . ;
```

From now on typing FRED followed by <ENTER> will print the square of the number on the top of the stack. For example,

4 FRED will print 16 .
−3 FRED will print 9.

If you want White Lightning to erase FRED from its memory, type FORGET FRED, but note that this also forgets any words defined before FRED.

Control structures now become increasingly important. A DO ... LOOP is very similar to a FOR ... NEXT loop in BASIC. For example,

```
: HELLO 10 1 DO I / LOOP ;
```

This defines a word to print the numbers from one to ten. The word 'I' puts the loop index on to the stack.

The next important concept is that of flags and conditions. There are several commands which test the numbers on the stack and replace them with a flag. A flag is either 0 for false or 1 for true. If one of these words is carried out then an IF structure usually follows. For example,

```
: TEST 0 < IF ." NEG" ELSE . POS" ENDIF ;
```

This will print "NEG" if the number on top of the stack is negative, otherwise it will print "POS". The instructions after the IF are carried out if the flag is true, but if it is false the words after ELSE are executed. If there are more than two possible actions then a similar CASE structure is implemented. The BEGIN ... UNTIL and BEGIN ... WHILE ... REPEAT loops operate on the same principle except that a flag is tested on receipt of the WHILE or UNTIL instructions and depending on the result it will either cause a jump out of the loop or a continuation of the loop.

To define a constant you must type the value, CONSTANT, and finally the name. For example,

```
3 CONSTANT PI
```

will assign the value 3 to the constant PI, which can now be used in exactly the same way as a number.

To define a variable type the value, VARIABLE, and the name. For example,

```
0 VARIABLE N
```

will assign the value 0 to the variable N. To put the value of N on top of the stack, type: 'N2.'. To change the value of N to 2, type: '2 N !'.

The editor is very easy to use and is extremely user-friendly. While programming in FORTH do not lose sight of p. 35 of the manual, which gives the meanings of the error messages. For a more extensive discussion of FORTH, consult Chapter 2 of this book.

IDEAL is a powerful animation sub-language, and you will need to spend time and be patient if you are to achieve the best results. The language is based on various types of windows and sprites. Screen windows are defined using the variables ROW, COL, HGT and LEN. Sprite windows are defined by SROW, SCOL, HGT and LEN. The sprites are stored between SPST and SPND, and unless you know what you are doing it is unwise to alter these two values. There are many different types of commands, but they can be divided into two groups:

1. Operations involving areas of the screen. These are post-fixed with 'S'.
2. Operations involving the sprites in memory. These are post-fixed with 'M'.

There are various scroll instructions which are listed on p. 119 and 120 of the manual. Vertical scrolls are controlled by the variable NPX, which stipulates the number of pixels to be scrolled. Those instructions that are post-fixed with 'V' affect the present screen window. There are three types of horizontal scrolls in each direction. In fact all the commands that could be implemented in the sprite generator have equivalents in IDEAL. There are many different types of GET and PUT commands, and you should already by fairly familiar with these.

The interrupt routine provides a very powerful background facility. The words which deal with the background are EXX, INT-ON, INT-OFF. Before you attempt to use the interrupts, check that you have not eradicated any of the 'garbage' on screens 1 to 5 of the listing, because this is where the interrupt machine code is stored. If you have then you will have to reload White Lightning.

By using standard FORTH words and variables it is possible to alter the frequency and phase of an interrupt. The background could be a scrolling landscape, as in Defender, or some moving extra men, as in Jet Set Willy.

Collision detection is amply provided for in the SCAN commands, which can be manipulated to detect the presence of

any pattern or a specific pattern. Another important feature of IDEAL is the BASIC interface, and all the possible requirements are met with the instructions PROG, RESERVE, GOTO and RETWR. Several different entry points are given so that a safe return from BASIC can easily be made. There are also all the BASIC hi-res commands to consider, and a list of these is given on p. 66 of the manual. Another important instruction is ZAP (or ZAP-INT) which produces a run time version of the present dictionary and prints its length. To save this final version, exit to BASIC using PROG and type SAVE "filename" CODE 24832,length. Programs in this form can be sold with no restrictions. Happy programming!

Bibliography

This book is an introduction to all the alternative languages available for the ZX-Spectrum, and accordingly no language has been considered in exhaustive detail. If you are interested in further information, the following books are highly recommended.

C-language

Thomas Plum, *Learning to Program in C*, Prentice Hall, 1983
Kernighan and Ritchie, *The C Programming Language*, Prentice Hall, 1978

FORTH

Alan Winfield, *The Complete FORTH*, Sigma Technical Press, 1983
Richard Harrison, *FORTH on the BBC Microprocessor*, Acornsoft, 1983
Owen Bishop, *Exploring FORTH*, Granada, 1984

LOGO

Peter Ross, *LOGO Programming*, Addison-Wesley, 1983
Boris Allan, *Introducing LOGO*, Granada, 1984

Pascal

Findlay and Watt, *Pascal: an introduction to methodical programming*, Pitman, 1980
Boris Allan, *Introducing Pascal*, Granada, 1984

Duckworth Home Computing

ALTERNATIVE LANGUAGES FOR THE SPECTRUM

Richard Hurley & David Virgo

Many Spectrum owners have appreciated the limitations of BASIC and are interested in experimenting with other high-level languages. These alternative languages are far more efficient for many applications and can be used to produce software of a more professional standard than BASIC without having to resort to the rigours of machine code.

This book is very machine-dependent, placing special emphasis on how each language can be used on the Spectrum to best effect. Chapters on LOGO, FORTH, PROLOG, PASCAL, PILOT and 'C' each consist of an introduction, a detailed description of the language, examples and ideas for future development. The final chapter looks at some highly acclaimed program development packages and discusses how these can be used in specific areas.

Richard Hurley is Head of Computer Studies at Hurstpierpoint College in Sussex, and has written several books on computing. David Virgo also teaches computing at Hurstpierpoint.

ISBN 0-7156-1978-0

9 780715 619780

Duckworth
The Old Piano Factory
43 Gloucester Crescent, London NW1

ISBN 0 7156 1978 0

IN UK ONLY £6.95 NET